Marc Ebner Michael O'Neill
Anikó Ekárt Leonardo Vanneschi
Anna Isabel Esparcia-Alcázar (Eds.)

LNCS 4445

Genetic Programming

10th European Conference, EuroGP 2007
Valencia, Spain, April 2007
Proceedings

Preface

In 2007 the European Conference on Genetic Programming (EuroGP) reached in its tenth year. What started out as a small workshop in 1998 in Paris has grown considerably in size over the years both in number of attendees as well as number of submissions. EuroGP is the only conference worldwide devoted exclusively to genetic programming and all aspects of evolutionary generation of computer programs. For the tenth year we came together to exchange our ideas on the automatic generation of programs inspired by Darwinian evolution. The main operators are reproduction, variation and selection. In nature, heritable traits are passed from one generation to the next. Variations are introduced through accidental mutations or by recombining genetic material from parents. Selection occurs as a result of limited resources. The very same process is used when trying to evolve programs using artificial evolution. The desired task for the programs to perform is specified via the fitness function.

This year we received a record number of 71 submissions. A rigorous, double-blind, selection mechanism was applied to the submitted papers. We accepted 21 plenary talks (30% acceptance rate) and 14 poster presentations (49% global acceptance rate for talks and posters). Each submissions was reviewed by at least three members of the international Program Committee from 19 different countries. Each reviewer was asked for keywords specifying their own area of expertise. Submissions were then appropriately matched to the reviewers based on their expertise using the optimal assignment of the conference management software (MyReview) originally developed by Philippe Rigaux, Bertrand Chardon and colleagues from the Université Paris-Sud Orsay, France. This version of the MyReview system has been developed with funding from the European Coordinated Action ONCE-CS (Open Network Connecting Excellence in Complex Systems), funded under FP6 framework by the FET division (contract 15539). Only small adjustments were then made manually to balance the work load better.

Papers were accepted for presentation at the conference based on the reviewers' recommendations and also taking into account originality, significance of results, clarity of representation, replicability, writing style, bibliography and relevance to the conference. As a result, 35 high-quality papers are included in these proceedings which address fundamental and theoretical issues such as crossover bias, issues such as chess game playing, real-time evaluation of VoIP, multi-objective optimization, evolution of recursive sorting algorithms, density estimation for inverse problem solving, image filter evolution, predicting prime numbers, data mining, grammatical genetic programming, layered learning, expression simplification, neutrality and evolvability, iterated function systems, particle swarm optimization, or open-ended evolution. The use of genetic

programming for several different applications shows that the method is a very general problem-solving paradigm.

The 10th European Conference on Genetic Programming took place during April 2007 11–13 in Valencia, Spain. The present volume contains all contributions that were accepted for publication either as talks or posters. All previous proceedings have been published by Springer in the *Lecture Notes in Computer Science* series. EuroGP was co-located with EvoCOP 2007, the seventh European Conference on Evolutionary Computation in Combinatorial Optimization, and also EvoBIO, fifth European Conference on Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics, and the series of EvoWorkshops, focusing on applications of evolutionary computation. Evo* (pronounced EvoStar) is the new umbrella name for the three co-located conferences and the EvoWorkshops series, the increasingly important international event exclusively dedicated to all aspects of evolutionary computing.

Many people helped to make the conference a success. We would like to express our gratitude to the members of the Program Committee for their thorough reviews of all papers submitted to the conference. Their constructive comments made it possible for the authors to improve their original submissions for final publication. We also thank the following institutions. The Universidad Politécnica de Valéncia provided institutional and financial support, as well as the lending of their premises and also helped with the organization and administration. The Instituto Tecnológico de Informática cooperated with regard to the local organization. The Spanish Ministerio de Educación y Ciencia also provided financial support for which we are very grateful. We are thankful to Marc Schoenauer, INRIA, France, for managing the MyReview conference management software.

We especially thank Jennifer Willies and the School of Computing, Napier University. Without her dedicated work and continued involvement with the EuroGP conference from the initial start in 1998 to what now has become Evo*, this event would not be what it is today.

April 2007

Marc Ebner
Michael O'Neill
Anikó Ekárt
Leonardo Vanneschi
Anna Isabel Esparcia-Alcázar

Organization

Administrative details were handled by Jennifer Willies, Napier University, School of Computing, Scotland, UK.

Organizing Committee

Program Co-chairs	Marc Ebner (Universität Würzburg, Germany)
	Michael O'Neill (University College Dublin, Ireland)
Publication Chair	Anikó Ekárt (Aston University, UK)
Local Chair	Anna Isabel Esparcia-Alcázar (Universidad Politécnica de Valencia, Spain)
Publicity Chair	Leonardo Vanneschi (University of Milano-Bicocca, Italy)

Program Committee

Abbass, Hussein. UNSW@ADFA, Australia
Altenberg, Lee. University of Hawaii at Manoa, USA
Aydin, Mehmet Emin. University of Bedfordshire, UK
Azad, R. Muhammad Atif. University of Limerick, Ireland
Banzhaf, Wolfgang. Memorial University of Newfoundland, Canada
Bentley, Peter. University College London, UK
Brabazon, Anthony. University College Dublin, Ireland
Bredeche, Nicolas. Université Paris-Sud, France
Burke, Edmund Kieran. University of Nottingham, UK
Cagnoni, Stefano. University of Parma, Italy
Cheang, Sin Man. Hong Kong Institute of Vocational Education, China
Collard, Philippe. I3S laboratory-UNSA, France
Collet, Pierre. Université du Littoral, Côte d'Opale, France
Costa, Ernesto. University of Coimbra, Portugal
de Jong, Edwin. Utrecht University, The Netherlands
Defoin Platel, Michael. Laboratoire d'Océanographie de Villefranche, France
Dempsey, Ian. University College Dublin, Ireland
Divina, Federico. Tilburg University, The Netherlands
Ebner, Marc. Universität Würzburg, Germany
Ekárt, Anikó. Aston University, UK
Essam, Daryl. University of New South Wales, Australia
Fernández de Vega, Francisco. University of Extremadura, Spain
Folino, Gianluigi. ICAR-CNR, Italy
Fonlupt, Cyril. LIL - Université du Littoral, Côte d'Opale, France
Gagné, Christian. Informatique WGZ Inc., Canada
Giacobini, Mario. University of Turin, Italy

Gustafson, Steven. GE Global Research, USA
Hao, Jin-Kao. University of Angers, France
Harvey, Inman. University of Sussex, UK
Hoai, Nguyen Xuan. The Vietnamese Military Technical Academy, Vietnam
Hornby, Greg. University of California, Santa Cruz, USA
Howard, Daniel. QinetiQ, UK
Johnson, Colin. University of Kent, UK
Kalganova, Tatiana. Brunel University, UK
Keijzer, Maarten. Chordiant Software, The Netherlands
Keller, Robert E. University of Essex, UK
Kendall, Graham. University of Nottingham, UK
Khan, Asifullah. Pakistan Inst. of Engineering and Applied Sciences, Pakistan
Kim, Daeeun. University of Leicester, UK
Kubalik, Jiri. Czech Technical University, Czech Republic
Levine, John. University of Strathclyde, UK
Lopes, Heitor Silverio. Federal Technological University of Parana, Brazil
Lucas, Simon. University of Essex, UK
Machado, Penousal. University of Coimbra, Portugal
Martin, Peter. Naiad Consulting Limited, UK
McKay, Bob. Seoul National University, Korea
Mehnen, Jörn. University of Dortmund, Germany
Miller, Julian. University of York, UK
Nicolau, Miguel. INRIA, France
Nievola, Julio Cesar. PUCPR, Brazil
O'Neill, Michael. University College Dublin, Ireland
O'Reilly, Una-May. Massachusetts Institute of Technology, USA
Pizzuti, Clara. ICAR-CNR, Italy
Poli, Riccardo. University of Essex, UK
Ray, Thomas. University of Oklahoma, USA
Robilliard, Denis. Université du Littoral, Cote d'Opale, France
Schoenauer, Marc. INRIA, France
Sekanina, Lukás. Brno University of Technology, Czech Republic
Sipper, Moshe. Ben-Gurion University, Israel
Skourikhine, Alexei. Los Alamos National Laboratory, USA
Soule, Terence. University of Idaho, USA
Tettamanzi, Andrea. University of Milan, Italy
Thompson, Adrian. University of Sussex, UK
Tomassini, Marco. University of Lausanne, Switzerland
van Hemert, Jano. University of Edinburgh, UK
Vanneschi, Leonardo. University of Milano-Bicocca, Italy
Verel, Sébastien. University of Nice-Sophia Antipolis, France
Yu, Tina. Memorial University of Newfoundland, Canada

Table of Contents

A Grammatical Genetic Programming Approach to Modularity in Genetic Algorithms . . . 1
Erik Hemberg, Conor Gilligan, Michael O'Neill, and Anthony Brabazon

An Empirical Boosting Scheme for ROC-Based Genetic Programming Classifiers . . . 12
Denis Robilliard, Virginie Marion-Poty, Sébastien Mahler, and Cyril Fonlupt

Confidence Intervals for Computational Effort Comparisons . . . 23
Matthew Walker, Howard Edwards, and Chris Messom

Crossover Bias in Genetic Programming . . . 33
Maarten Keijzer and James Foster

Density Estimation with Genetic Programming for Inverse Problem Solving . . . 45
Michael Defoin Platel, Sébastien Vérel, Manuel Clergue, and Malik Chami

Empirical Analysis of GP Tree-Fragments . . . 55
Will Smart, Peter Andreae, and Mengjie Zhang

Empirical Comparison of Evolutionary Representations of the Inverse Problem for Iterated Function Systems . . . 68
Anargyros Sarafopoulos and Bernard Buxton

Evolution of an Efficient Search Algorithm for the Mate-In-N Problem in Chess . . . 78
Ami Hauptman and Moshe Sipper

Fast Genetic Programming on GPUs . . . 90
Simon Harding and Wolfgang Banzhaf

FIFTH™: A Stack Based GP Language for Vector Processing . . . 102
Kenneth Holladay, Kay Robbins, and Jeffery von Ronne

Genetic Programming with Fitness Based on Model Checking . . . 114
Colin G. Johnson

Geometric Particle Swarm Optimisation . . . 125
Alberto Moraglio, Cecilia Di Chio, and Riccardo Poli

GP Classifier Problem Decomposition Using First-Price and Second-Price Auctions 137
Peter Lichodzijewski and Malcolm I. Heywood

Layered Learning in Boolean GP Problems 148
David Jackson and Adrian P. Gibbons

Mining Distributed Evolving Data Streams Using Fractal GP Ensembles 160
Gianluigi Folino, Clara Pizzuti, and Giandomenico Spezzano

Multi-objective Genetic Programming for Improving the Performance of TCP 170
Cyril Fillon and Alberto Bartoli

On Population Size and Neutrality: Facilitating the Evolution of Evolvability 181
Richard M. Downing

On the Limiting Distribution of Program Sizes in Tree-Based Genetic Programming 193
Riccardo Poli, William B. Langdon, and Stephen Dignum

Predicting Prime Numbers Using Cartesian Genetic Programming 205
James Alfred Walker and Julian Francis Miller

Real-Time, Non-intrusive Evaluation of VoIP 217
Adil Raja, R. Muhammad Atif Azad, Colin Flanagan, and Conor Ryan

Training Binary GP Classifiers Efficiently: A Pareto-coevolutionary Approach 229
Michal Lemczyk and Malcolm I. Heywood

Posters

A Comprehensive View of Fitness Landscapes with Neutrality and Fitness Clouds 241
Leonardo Vanneschi, Marco Tomassini, Philippe Collard, Sébastien Vérel, Yuri Pirola, and Giancarlo Mauri

Analysing the Regularity of Genomes Using Compression and Expression Simplification 251
Jungseok Shin, Moonyoung Kang, R.I. (Bob) McKay, Xuan Nguyen, Tuan-Hao Hoang, Naoki Mori, and Daryl Essam

Changing the Genospace: Solving GA Problems with Cartesian Genetic Programming 261
James Alfred Walker and Julian Francis Miller

Code Regulation in Open Ended Evolution 271
Lidia Yamamoto

Data Mining of Genetic Programming Run Logs 281
Vic Ciesielski and Xiang Li

Evolving a Statistics Class Using Object Oriented Evolutionary Programming 291
Alexandros Agapitos and Simon M. Lucas

Evolving Modular Recursive Sorting Algorithms 301
Alexandros Agapitos and Simon M. Lucas

Fitness Landscape Analysis and Image Filter Evolution Using Functional-Level CGP 311
Karel Slaný and Lukáš Sekanina

Genetic Programming Heuristics for Multiple Machine Scheduling 321
Domagoj Jakobović, Leonardo Jelenković, and Leo Budin

Group-Foraging with Particle Swarms and Genetic Programming 331
Cecilia Di Chio and Paolo Di Chio

Multiple Interactive Outputs in a Single Tree: An Empirical Investigation 341
Edgar Galván-López and Katya Rodríguez-Vázquez

Parsimony Doesn't Mean Simplicity: Genetic Programming for Inductive Inference on Noisy Data 351
Ivanoe De Falco, Antonio Della Cioppa, Domenico Maisto, Umberto Scafuri, and Ernesto Tarantino

The Holland Broadcast Language and the Modeling of Biochemical Networks 361
James Decraene, George G. Mitchell, Barry McMullin, and Ciaran Kelly

The Induction of Finite Transducers Using Genetic Programming 371
Amashini Naidoo and Nelishia Pillay

Author Index 381

A Grammatical Genetic Programming Approach to Modularity in Genetic Algorithms

Erik Hemberg[1], Conor Gilligan[1], Michael O'Neill[1], and Anthony Brabazon[2]

[1] UCD Natural Computing Research & Applications
School of Computer Science and Informatics
University College Dublin, Ireland
erik.hemberg@ucd.ie, conor.gilligan@ucd.ie, m.oneill@ucd.ie
[2] UCD Natural Computing Research & Applications
School of Business
University College Dublin, Ireland
anthony.brabazon@ucd.ie

Abstract. The ability of Genetic Programming to scale to problems of increasing difficulty operates on the premise that it is possible to capture regularities that exist in a problem environment by decomposition of the problem into a hierarchy of modules. As computer scientists and more generally as humans we tend to adopt a similar divide-and-conquer strategy in our problem solving. In this paper we consider the adoption of such a strategy for Genetic Algorithms. By adopting a modular representation in a Genetic Algorithm we can make efficiency gains that enable superior scaling characteristics to problems of increasing size. We present a comparison of two modular Genetic Algorithms, one of which is a Grammatical Genetic Programming algorithm, the meta-Grammar Genetic Algorithm (mGGA), which generates binary string sentences instead of traditional GP trees. A number of problems instances are tackled which extend the Checkerboard problem by introducing different kinds of regularity and noise. The results demonstrate some limitations of the modular GA (MGA) representation and how the mGGA can overcome these. The mGGA shows improved scaling when compared the MGA.

1 Introduction

In the natural world examples of modularity and hierarchies abound, ranging the biological evolution of cells to form tissues and organs to the physical structure of matter from the sub-atomic level up. In most examples of problem solving by humans, regularities in the problem environment are exploited in a divide-and-conquer approach through the construction of sub-solutions, which may then be reused and combined in a hierarchical fashion to solve the problem as a whole. Similarly Genetic Programming provides as components of its problem solving toolkit the ability to automatically create, modify and delete modules, which can be used in a hierarchical fashion. The objectives of this study are to investigate the adoption of principles from Genetic Programming [1] such as modularity and reuse (see Chapter 16 in [2]) for application to Genetic Algorithms, and to

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 1–11, 2007.
© Springer-Verlag Berlin Heidelberg 2007

couple these to an adaptive representation that allows the type and usage of these principles to be evolved through the use of evolvable grammars. The goal being the development of an evolutionary algorithm with good scaling characteristics, and an adaptable representation that will facilitate its application to noisy, dynamic, problem environments. To this end a grammar-based Genetic Programming approach is adopted, in which the grammars represent the construction of syntactically correct genotypes of the Genetic Algorithm. In particular, we compare the representations and performance of the meta-Grammar Genetic Algorithm (mGGA) [3] to the Modular Genetic Algorithm (MGA) [4], highlighting some of the MGA's representational limitations, and demonstrate the potential of a more expressive representation in the form of the mGGA to scale to problems of increasing size and difficulty. Additionally, we consider the introduction of noise into the Checkerboard problem, in order to assess how the representations might generalise into noisy, real-world problem domains. The remainder of the paper is structured as follows. Section 2 provides background on earlier work in modular GAs and describes the meta-Grammar Genetic Algorithm. Section 3 details the experimental approach adopted and results, and finally section 4 details conclusions and future work.

2 Background

There has been a large body of research on modularity in Genetic Programming and effects on its scalability, however the same cannot be stated for the Genetic Algorithm (GA). In this section we present two modular representations as implemented in the Modular GA [4] and the meta-Grammar GA [3].

2.1 Modular Genetic Algorithm

Garibay et al. introduced the Modular Genetic Algorithm, which was shown to signficantly outperform a standard Genetic Algorithm on a scalable problem with regularities [4]. The genome of an MGA individual is a vector of genes, where each gene is comprised of two components, the `number-of-repetitions` and some `function` which is repeated according to the value of the repetitions field. For example, if we had a function (`one()`) that always returned the value `1` when called and another (`zero()`) that returned the value `0` we have a representation that can generate binary strings. A sample individual comprised of three genes might look like: {`2, zero()`}, {`4, one()`}, {`2, zero()`}, which would produce the binary string `00111100`. The MGA was shown to have superior ability to scale to problems of increasing complexity than a standard GA.

2.2 Grammatical Evolution by Grammatical Evolution

The grammar-based Genetic Programming approach upon which this study is based is the Grammatical Evolution by Grammatical Evolution algorithm [5], which is in turn based on the Grammatical Evolution algorithm [6,7,8,9]. This is

a meta-Grammar Evolutionary Algorithm in which the input grammar is used to specify the construction of another syntactically correct grammar. The generated grammar is then used in a mapping process to construct a solution. In order to allow evolution of a grammar (Grammatical Evolution by Grammatical Evolution $(GE)^2$), we must provide a grammar to specify the form a grammar can take. This is an example of the richness of the expressiveness of grammars that makes the GE approach so powerful. See [6,10,11] for further examples of what can be represented with grammars and [12] for an alternative approach to grammar evolution. By allowing an Evolutionary Algorithm to adapt its representation (in this case through the evolution of the grammar) it provides the population with enhanced robustness in the face of a dynamic environment, in particular, and also to automatically incorporate biases into the search process. In this case we can allow the meta-Grammar Genetic Algorithm to evolve biases towards different building blocks of varying sizes. In this approach we therefore have two distinct grammars, the *universal grammar* (or grammars' grammar) and the *solution grammar*. The notion of a universal grammar is adopted from linguistics and refers to a universal set of syntactic rules that hold for spoken languages [13]. It has been proposed that during a child's development the universal grammar undergoes modifications through learning that allows the development of communication in their parents native language(s) [14]. In $(GE)^2$ the universal grammar dictates the construction of the solution grammar. In this study two separate, variable-length, genotypic binary chromosomes were used, the first chromosome to generate the solution grammar from the universal grammar and the second chromosome generates the solution itself. Crossover operates between homologous chromosomes, that is, the solution grammar chromosome from the first parent recombines with the solution grammar chromosome from the second parent, with the same occurring for the solution chromosomes. In order for evolution to be successful it must co-evolve both the meta-Grammar and the structure of solutions based on the evolved meta-Grammar, and as such the search space is larger than in standard Grammatical Evolution.

2.3 Meta-grammars for Bitstrings

A simple grammar for a fixed-length (8 bits in the following example) binary string individual of a Genetic Algorithm is provided below. In the generative grammar each bit position (denoted as `<bit>`) can become either of the boolean values. A standard variable-length Grammatical Evolution individual can then be allowed to specify what each bit value will be by selecting the appropriate `<bit>` production rule for each position in the `<bitstring>`.

```
<bitstring> ::= <bit><bit><bit><bit><bit><bit><bit><bit>
<bit> ::= 1 | 0
```

The above grammar can be extended to incorporate the reuse of groups of bits (building blocks). In this example all building blocks that are mutliples of two are provided, although it would be possible to create a grammar that adopted more complex arrangements of building blocks.

```
<bitstring> ::= <bbk4><bbk4> | <bbk2><bbk2><bbk2><bbk2>
              | <bbk1><bbk1><bbk1t><bbk1><bbk1><bbk1><bbk1><bbk1>
<bbk4> ::= <bit><bit><bit><bit>
<bbk2> ::= <bit><bit>
<bbk1> ::= <bit>
<bit> ::= 1 | 0
```

The above grammars are static, and as such can only allow one building block of size four and of size two in the second example. It would be better to allow our search algorithm the potential to uncover a number of building blocks of any one size from which a Grammatical Evolution individual could choose from. This would facilitate the application of such a Grammatical GA to:

- problems with more than one building block type for each building block size,
- to search on one building block while maintaining a *reasonable* temporary building block solution,
- and to be able to switch between building blocks in the case of dynamic environments.

All of this can be achieved through the adoption of meta-Grammars as were adopted earlier in [5]. An example of such a grammar for an 8-bit individual is given below.

```
<g> ::=  "<bitstring> ::=" <reps>
              "<bbk4> ::=" <bbk4t>
              "<bbk2> ::=" <bbk2t>
              "<bbk1> ::=" <bbk1t>
               "<bit> ::=" <val>

<bbk4t> ::= <bit><bit><bit><bit>
<bbk2t> ::= <bit><bit>
<bbk1t> ::= <bit>
<reps> ::= <rept> | <rept> "|" <reps>
<rept> ::= "<bbk4><bbk4>" | "<bbk2><bbk2><bbk2><bbk2>"
         | "<bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1>"
<bit> ::= "<bit>" | 1 | 0
<val> ::= <valt>  | <valt> "|" <val>
<valt> ::= 1 | 0
```

In this case the grammar specifies the construction of another generative bitstring grammar. The subsequent bitstring grammar that can be produced from the above meta-grammar is restricted such that it can contain building blocks of size 8. Some of the bits of the building blocks can be fully specified as a boolean value or may be left as unfilled for the second step in the mapping process. An example bitstring grammar produced from the above meta-grammar could be:

```
<bitstring> ::= <bit>11<bit>00<bit><bit> | <bbk2><bbk2><bbk2><bbk2>
              | 11011101 | <bbk4><bbk4> | <bbk4><bbk4>
<bbk4> ::= <bit>11<bit>
<bbk2> ::= 11
<bbk1> ::= 1
<bit> ::= 1 | 0 | 0 | 1
```

To allow the creation of multiple building blocks of different sizes the following grammar could be adopted (again shown for 8-bit strings).

```
<g> ::=  "<bitstring> ::=" <reps>
              "<bbk4> ::=" <bbk4>
              "<bbk2> ::=" <bbk2>
              "<bbk1> ::=" <bbk1>
               "<bit> ::=" <val>

<bbk4> ::= <bbk4t> | <bbk4t> "|" <bbk4>
<bbk2> ::= <bbk2t> | <bbk2t> "|" <bbk2>
<bbk1> ::= <bbk1t> | <bbk1t> "|" <bbk1>
<bbk4t> ::= <bit><bit><bit><bit>
<bbk2t> ::= <bit><bit>
<bbk1t> ::= <bit>
<reps> ::= <rept> | <rept>  "|" <reps>
<rept> ::= "<bbk4><bbk4>" | "<bbk2><bbk2><bbk2><bbk2>"
         | "<bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1>"
<bit> ::= "<bit>" | 1 | 0
<val> ::= <valt> | <valt> "|" <val>
<valt> ::= 1 | 0
```

An example bitstring grammar produced by the above meta-grammar is provided below.

```
<bitstring> ::= <bit>11<bit>00<bit><bit> | <bbk2><bbk2><bbk2><bbk2>
              | 11011101 | <bbk4><bbk4> | <bbk4><bbk4>
<bbk4> ::= <bit>11<bit> | 000<bit>
<bbk2> ::= 11 | 00 | <bit>1
<bbk1> ::= 0 | 0
<bit> ::= 1 | 0 | 0 | 1
```

Modularity exists above in the ability to specify the size and content (or partial content) of a buiding block through its incorporation into the solution grammar. This building block can then be repeatedly reused in the generation of the phenotype. An additional mechanism for reuse is through the Wrapping operator of Grammatical Evolution. During the mapping process if we reach the end of the genotype and still have outstanding decisions to make on the construction of our phenotype we can invoke the wrapping operator to move our reading head back to the first codon in the genome. This allows the reuse of rule choices if the codons are used in the same context. Given that the lengths of binary strings which may need to be represented can grow quite large it is possible to automate the creation of meta-grammars by simply providing the length of the target solution and creating all possible building block structures that can be used to create a bitstring of the target length. In this study the target binary strings are of lengths 60, 90, 120, 180, and 210. The building block sizes incorporated in their corresponding grammars are therefore all integers that divide into the target string lengths (i.e., for a target string of length 60 the building blocks are of sizes 30, 20, 15, 12, 10, 6, 5, 4, 3, 2 and 1). Meta-grammars are of course not limited to the specification of grammars for binary strings and can be easily extended to the representation of real and integer strings as well as programs, or any structure which can be represented in a grammatical form.

3 Experimental Setup and Results

Before detailing the experimental design and setup we first introduce the problems on which we will benchmark the two representations under investigation.

3.1 The Checkerboard-Pattern Discovery Problem

Given the lack of suitable benchmark problems in the Genetic Algorithm literature that consider modularity, Garibay et al., [4] proposed the Checkerboard-Pattern discovery problem. In this problem a pattern of colours or states is imposed upon a two dimensional grid called the Checkerboard. There are 2 possible states adopted for each square on the grid, i.e., black or white, which can be represented as bit values 1 and 0 respectively. Each candidate solution tries to recapture the pattern contained in the target Checkerboard. Fitness is simply measured by summing the number of squares that contain the correct state. In this study we then normalise fitness to the range 0.0 to 1.0, and standardise fitness such that 0.0 is the best possible fitness where all of the candidate solution's squares exactly match the target checkerboard-pattern. It is easily possible to scale the problem in terms of its complexity, modularity and regularity by increasing the size of the checkerboard, the number of patterns, and changing the number of components in each pattern, respectively. Example instances of this problem which are adopted in this study and in [4] are presented in Fig.1, which illustrates scaled-up versions of a 4X8 pattern to 8X16 and 16X32. Another problem instance tackled in this study of a 8X16 checkerboard pattern is also illustrated. A third set of problem instances are examined which add noise to the state of each sqaure upon the evaluation of each individual. This is implemented by randomly switching the state of a square with a predefined probability for the patterns already presented in Fig.1. With the addition of noise to the regular patterns this makes it more challenging to uncover the underlying patterns and thus add an additional element of real-world interest to this benchmark problem. The amount of noise can easily be tuned by altering the probability of error.

Table 1. Performance changes for the mGGA on the standard non-noisy problem instances. The average best fitness after 500 generations is 0.019792 for 2^{4X8} and after a 1000 generations 0.019531 for 2^{8X16}. The differnece in fitness between the two generations is 0.000261. The average best fitness after 400 generations for 2^{16X32} is 0.01875. Difference between 2^{4X8} and 2^{16X32} is 0.001042.

	Performance drop (% of fitness decrease)	
Complexity increase	MGA	mGGA
from 2^{4X8} to 2^{8X16}	3.68%	0.02%
from 2^{4X8} to 2^{16X32}	11.38%	0.1%

3.2 Comparing Performance of mGGA and MGA

30 runs on each problem instance were performed with the mGGA using a population size of 1000, tournament selection (size 3), mutation probability of 0.001 per gene, and crossover probability of 0.7. The number of generations was selected to reflect the values adopted in Garibay et al's study [4], i.e. 500 for the 4X8, 1000 for the 8X16 and 2000 generations for the 16X32 problem

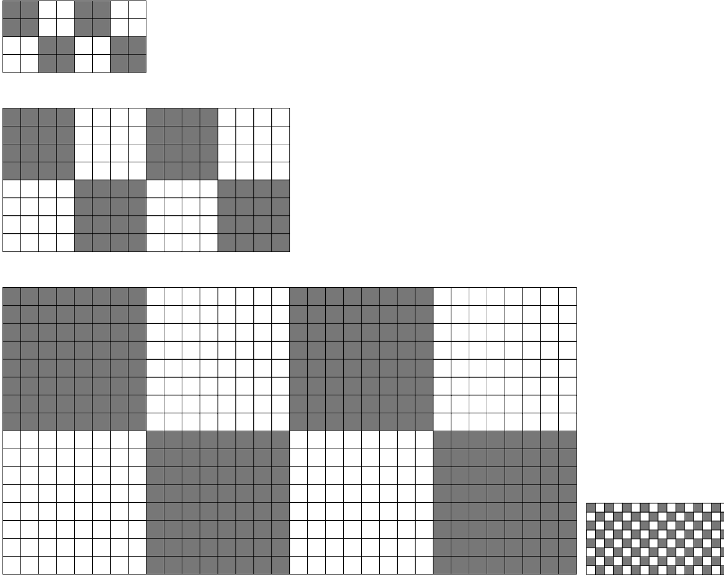


Fig. 1. The original checkerboard-pattern matching problem instances (from left 2^{4X8}, 2^{8X16} and 2^{16X32}) as presented in [4]. On the far right is a new 2^{8X16} checkerboard-pattern matching problem checkerboard instance with finer-grained regularity.

Table 2. Performance values for the mGGA on the standard non-noisy problem instances. Average values are for 30 runs. The value in parenthsis is random search for 1000000 tries.

Problem	Best fitness	Mean fitness	Variance(best fit.)	Successful Runs
2^{4X8}	0.0119	0.0168	0.0017	26/30
2^{8X16}	0.0211 (0.164)	0.0265 (0.25)	0.0030	25/30
2^{16X32}	0.0188 (0.416)	0.0034 (0.5)	0.0248	27/30

instance. Random initialisation was used, and the fitness values and their variance reported in the initial population averaged over the 30 runs reflects this (see e.g. Fig. 2). The results are reported in Fig 2, with the percentage gains in performance and fitness statistics reported in Tables 1 and 2 respectively. It is clear that as the problem instances increase in complexity there are economies of scale to be achieved, with the relative performance of the mGGA improving significantly with each jump in problem size. An additional problem instance as portrayed in Fig.1(far right) was examined with the same parameters, with the results presented in Fig.3. In this case the pattern is of size 1X1 and as such is much finer grained than the patterns examined earlier. It is difficult for the MGA to efficiently represent a solution to this problem instance due to the nature of the pattern. Effectively each squares state must be specified individualy. However,

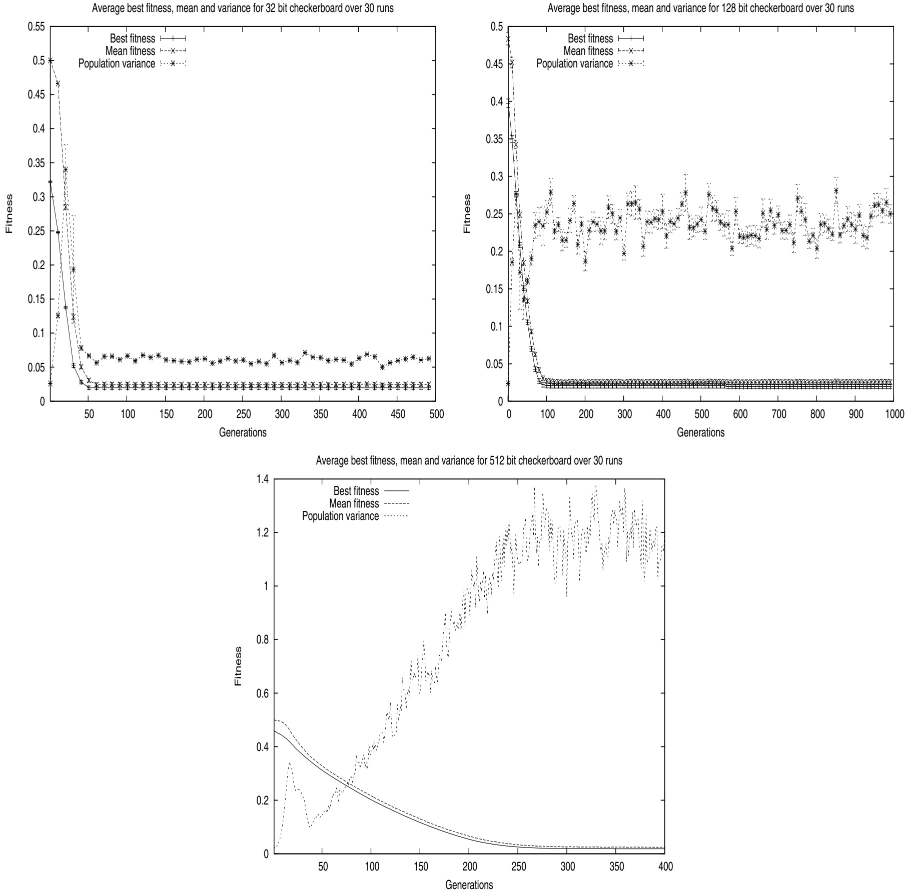


Fig. 2. A graph for the mGGA on 2^{4X8} (top left), 2^{8X16} (top right), and 2^{8X32} instances (bottom)

this is not the case with the mGGA which can encode effectively parameterise the evolved modules to specify multiple square states with different values.

3.3 mGGA Performance Under Noisy Conditions

In order to gain some preliminary insight into the performance of the mGGA in a more realistic real-world setting it was decided to conduct experimental runs incorporating noise into the target patterns. This was achieved by simply flipping each bit in the target pattern with probability p_n. Runs were conducted using the same parameters as previously described for noise probabilities $p_n = 0.05, 0.075$ on the 2^{128} sized problem. The results achieved here are presented in Table 3 and Fig.4. As can be expected, the addition of noise reduced the algorithm performance on average, however on inspection of individual runs it was seen that this

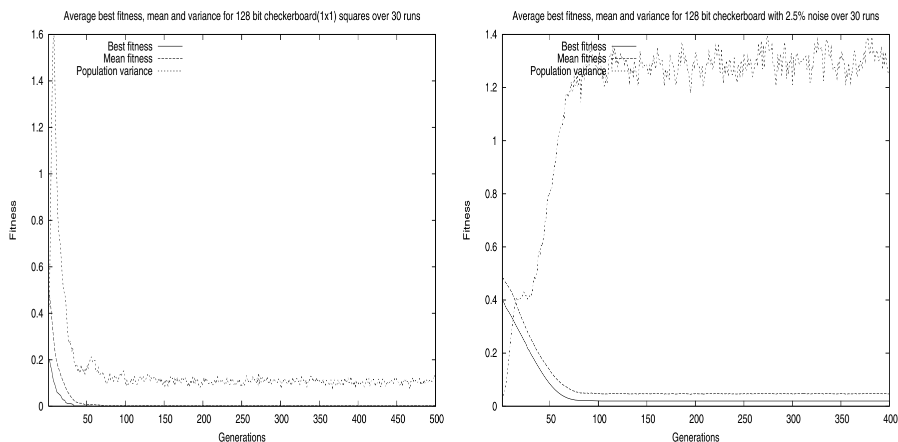


Fig. 3. A graph for the mGGA on 2^{8X16} checkerboard (1X1) (left) and, on the right the standard 2^{8X16} instance with 2.5% noise

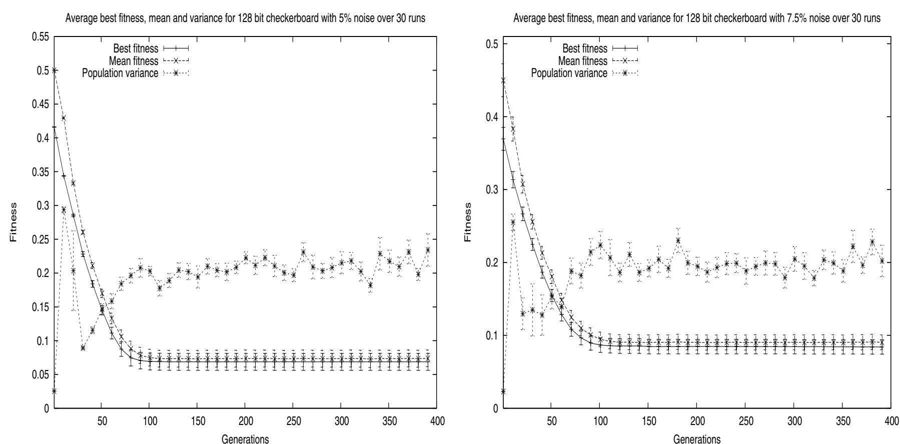


Fig. 4. A graph for the mGGA on 2^{8X16} with 5% noise (left) and the 2^{8X16} with 7.5% noise (right)

performance drop was manifest in an increased, but still small, number of runs which failed to converge to an optimal solution; instead converging prematurely on areas of very poor fitness. This indicates that the population may be converging too quickly in the early stages of the algorithm, loosing whatever diversity was present in the initial population. It is possible that through adjusting parameters of the EA better results could be achieved. It may also be wise to examine the initialization technique as it is possible that the initial population lacks diversity. The increase in search space size with the use of both meta-grammar and solution chromosomes may also be having an impact on performance.

Table 3. Statistics for performance of the mGGA on the Noisy Checkerboard-Pattern Discovery instances

Noise level	Best fitness	Mean fitness	Variance(best fit.)	Successful Runs
$S(1x1)$ $p = 0$	0	0.0024	0	30/30
$p = 0$	0.0195	0.0253	0.0027	25/30
$p = 0.025$	0.0198	0.0468	0.002	24/30
$p = 0.05$	0.0664	0.0719	0.0123	22/30
$p = 0.075$	0.0841	0.0904	0.0098	13/30

4 Conclusions and Future Work

We presented a comparison of the meta-Grammar GA (mGGA) to the Modular GA (MGA), illustrating the application of evolvable grammars to implement modularity in Genetic Algorithms. We also introduced a number of variations to the benchmark Checkerboard-Pattern discovery problem including different types of regularity and the introduction of noise to bring the benchmark closer to real-world scenarios. On the problem instances examined there are clear performance advantages for the mGGA when compared to the MGA. In addition to the application to more benchmark problem instances in particular to those belonging to the dynamic class, future work will investigate the effects alternative grammars and comparisons to other GAs from the literature including the competent GAs. A number of avenues to facilitate the co-evolution of the grammar and solution, such as different operator probabilities, will also be investigated.

Acknowledgement

This research is based upon works supported by the Science Foundation Ireland under Grant No. 06/RFP/CMS042.

References

1. Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.
2. Koza, J. R., Keane, M. A., Streeter, M. J., Mydlowec, W., Yu, J., Lanza, G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers.
3. O'Neill M., Brabazon A. (2005). mGGA: The Meta-Grammar Genetic Algorithm. *LNCS 3447 Proceedings of the European Conference on Genetic Programming EuroGP 2005*, pp.311-320. Springer.
4. Garibay O.O., Garibay I.I., Wu A.S. (2003). The Modular Genetic Algorithm: Exploiting Regularities in the Problem Space. *In LNCS 2869 Computer and Information Science ISCIS 2003*, pp.584-591. Springer.

5. O'Neill, M., Ryan, C. (2004). Grammatical Evolution by Grammatical Evolution. The Evolution of Grammar and Genetic Code. *LNCS 3003. Proc. of the European Conference on Genetic Programming 2004*, pp. 138-149. Springer.
6. O'Neill, M., Ryan, C. (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers.
7. O'Neill, M. (2001). Automatic Programming in an Arbitrary Language: Evolving Programs in Grammatical Evolution. PhD thesis, University of Limerick.
8. O'Neill, M., Ryan, C. (2001). Grammatical Evolution, *IEEE Trans. Evolutionary Computation*. 2001.
9. Ryan, C., Collins, J.J., O'Neill, M. (1998). Grammatical Evolution: Evolving Programs for an Arbitrary Language.*Proc. of the First European Workshop on GP*, pp. 83-95, Springer-Verlag.
10. Dempsey, I., O'Neill, M., Brabazon, A. (2004). Grammatical Constant Creation. In LNCS 3103, *Proceedings of GECCO 2004*, Part 1, pp. 447-458, Seattle, USA.
11. O'Neill, M., Cleary, R. (2004). Solving Knapsack Problems with Attribute Grammars. In *Proceedings of the Grammatical Evolution Workshop 2004*, GECCO 2004, Seattle, USA.
12. Shan, Y., McKay, R. I., Baxter, R., Abbas, H., Essam, D., Nguyen, H.X. (2004). Grammar Model-based Program Evolution. In *Proceedings of the 2004 Congress on Evolutionary Computation, CEC 2004*, Vol. 1, pp. 478-485, Portland, USA.
13. Chomsky, N. (1975). *Reflections on Language*. Pantheon Books. New York.
14. Pinker, S. (1995). *The language instinct: the new science of language and the mind.* Penguin, 1995.

An Empirical Boosting Scheme for ROC-Based Genetic Programming Classifiers

Denis Robilliard, Virginie Marion-Poty, Sébastien Mahler, and Cyril Fonlupt

Laboratoire d'Informatique du Littoral,
Maison de la Recherche Blaise Pascal,
50 rue Ferdinand Buisson - BP 719, 62228 CALAIS Cedex, France
{robillia,poty,mahler,fonlupt}@lil.univ-littoral.fr

Abstract. The so-called 'boosting' principle was introduced by Schapire and Freund in the 1990s in relation to weak learners in the Probably Approximately Correct computational learning framework. Another practice that has developed in recent years consists in assessing the quality of evolutionary or genetic classifiers with Receiver Operating Characteristics (ROC) curves. Following the RankBoost algorithm by Freund et al., this article is a cross-bridge between these two techniques, and deals about boosting ROC-based genetic programming classifiers. Updating the weights after a boosting round turns to be the algorithm keystone since the ROC curve does not allow to know directly which training cases are learned or misclassified. We propose a geometrical interpretation of the ROC curve to attribute an error measure to every training case. We validate our ROCboost algorithm on several benchmarks from the UCI-Irvine repository, and we compare boosted Genetic Programming performance with published results on ROC-based Evolution Strategies and Support Vector Machines.

1 Introduction

This paper is a cross-bridge between two Machine Learning techniques, namely boosting and Receiver Operating Characteristics (ROC) based evolutionary learning. Its first founding stone is the idea of "boosting" i.e. combining the results of a set of so-called "weak" learners to obtain a powerful combined predictor, that originates from the work of Schapire and Freund in the 1990s [1,2,3]. Starting from weak learners as they are defined in the Probably Approximately Correct (PAC) learning framework, i.e. learners that do just a little better than random guess, Schapire and Freund proved that the boosting procedure can yield a strong learner able to come as close as desired to perfect precision, in theory. They also published the AdaBoost algorithm that was able to improve real learners, such as C4.5, that are not true weak learners. Moreover AdaBoost proved to be largely resistant, although not immune, to over-fitting in many practical cases. These achievements raised much interest in the Machine Learning community, the boosting procedure being even qualified as "best off-the-shelf classifier" by Leo Breiman [4]. An enlightening analysis of boosting has also been

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 12–22, 2007.
© Springer-Verlag Berlin Heidelberg 2007

produced by Friedman et al. in [5] where its relationship with additive logistic regression is deeply studied.

The main idea in boosting is to perform several consecutive learning phases, called rounds, on a training set where every sample is assigned a weight, all weights forming a distribution and being equal for the first boosting round. At the end of every round an error (or loss) is computed to derive a confidence value for the current round hypothesis, and misclassified samples get their weights increased in relation to this confidence. A new learning phase is then started to generate another hypothesis. Once all learning rounds are finished, we obtain a combined predictor through a confidence weighted majority vote of all hypotheses. This procedure is summed up in Table 1.

Table 1. Pseudo code for AdaBoost algorithm

---

let $S = \{(x_i, y_i)\}$, $i = 1, \ldots, n, x_i \in X, y_i \in \{1, -1\}$, be the training samples set.
initialize the weights $w_i = 1/n$, $i = 1, \ldots, n$
for r in 1..t // iterate t boosting rounds
  learn hypothesis $h_r(x) \to \{-1, 1\}$ using training samples weighted by w_i
  compute error $\epsilon_r = \Sigma_{i:h_r(x_i) \neq y_i} w_i$
  set confidence $c_r = \log((1 - \epsilon_r)/\epsilon_r)$
  update misclassified samples weights $w_{i:h_r(x_i) \neq y_i} = w_{i:h_r(x_i) \neq y_i} e^{c_r}$
  re-normalize so that $\Sigma_{i=1}^{n} w_i = 1$
end for
let $(x, ?)$ be a test sample with unknown class.
output combined predictor: $\text{sign}(\Sigma_{r=1}^{t} c_r h_r(x))$

The second founding stone of the paper is ROC-based evolutionary classifiers. ROC curves are popular for medical data analysis, as they offer an intuitive representation of the true positive to false positive compromise when setting a threshold level on a given characteristic to discriminate pathological cases (see illustration in Fig. 1). ROC curves analysis also allows to bypass the difficulties associated to predictive accuracy when dealing with highly biased distribution of positive versus negative samples, or with different costs between misclassified positive and negative cases. This is why several authors recommend to use the area under the ROC curve — also known as AUC — rather than predictive accuracy as a measure for comparing learning algorithm, see [6,7]. In this case, one tries to maximize the AUC, and since it is a NP-complete problem [8] several works rely on evolutionary computation to do the optimization job, such as Mozer et al. [9] or Sebag's et al. ROGER algorithm [10,11,12].

When choosing to implement a ROC-based learning scheme, one needs to opt for hypotheses associated to a ROC characteristic such that increasing a threshold on that characteristic yields a monotonous increase in the rate of both true positives and false positives. Typical examples are real-valued functions: let $h : X \to \mathbb{R}$ be a predictor on the instances features space X, h induces a set of

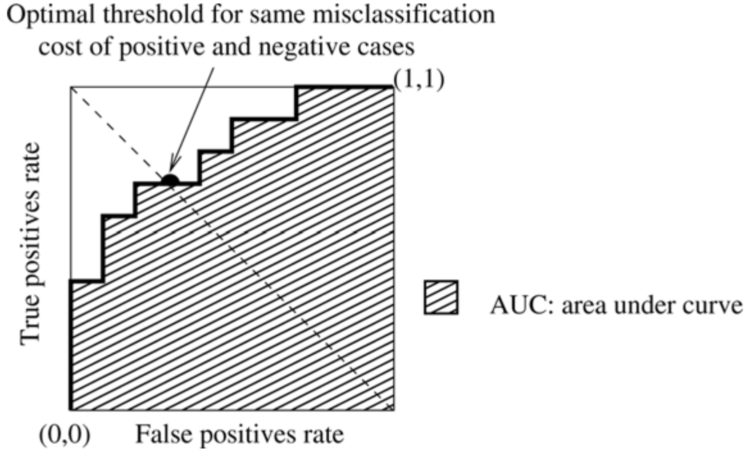


Fig. 1. Illustration of a ROC curve. The staircase effect is due to the curve being drawn from a finite number of training samples.

binary classifiers $\{h_t\}$ by using a real threshold value t such that: $h_t(x) = 1$ if $h(x) > t$ else $h_t(x) = -1$.

In this paper we propose to adapt boosting in order to enhance the performance of ROC-based Genetic Programming (GP) and Evolution Strategies (ES) learners. For this matter, the RankBoost algorithm from Freund et al. [13] is a boosting scheme dedicated to combine ranking hypotheses, i.e. hypotheses that try to order a set of samples. It has been shown to boost the AUC in [14], so it could suit our need, but it constrains the choice of the learner, that must either output values in $\{0, 1\}$ or in the $[0, 1]$ interval, or that requires a numerical search to optimize some internal parameters of the algorithm. We propose another boosting scheme that lifts these constraints and thus allows to directly re-use existing real-valued GP or ES classifiers. To accomplish this goal we need first to take into account the weights distribution into the evolutionary learning procedure, and then to design a weight update scheme. The easiest way to associate weights to the learning phase consists in performing a weight biased random re-sampling of the training cases as was done e.g. by Freund and Schapire for boosting C4.5 [3] or by Iba for boosting Genetic Programming [15]. As pointed out by [5] the effects of such re-sampling are still an open question, and one can rather consider to use weights inside the learning process. This is the choice we have made here, so the weights are used in the computation of the area under ROC curve, by slightly modifying the efficient $n * \log(n)$ algorithm given in [10]. The second point is the weights update phase, which is less obvious when dealing with hypotheses that yield a ROC curve, i.e. we cannot say if a training case is misclassified or not since this depends on the variable threshold to be put on the ROC characteristic. We rather propose to consider the area *above* the ROC curve as an error measure of the

hypothesis, to be partitioned and attributed to each sample case, taking into account its current weight. This is akin to the RankBoost algorithm but it differs in that our error measure is independent from the actual learner output (i.e. only the relative order of output values is used, not the difference between these values).

Weighted AUC and update procedures (that we call ROCboost for short) are explained in the next section, validation experiments on instances from the UCI-Irvine repository[1] are detailed in Section 3, then we conclude by a discussion on the results and drawbacks of the algorithm, and sketch future works.

2 Boosting Evolutionary ROC-Based Learner

In this section we give a detailed description of the two main components of the ROCboost algorithm. The weighted AUC computation pseudo code is shown in Table 2 and serves as a fitness function for the evolutionary learner. It is very close to the one proposed by Sebag et al. in their ROGER experiments [10], except that the contribution of every learning case to the global area under curve is proportional to its weight. Note that a part of the sorting criterion has been corrected from [10]: it should be $(y_i < y_j)$ because the area under curve should not increase when an hypothesis outputs the same real value for several positive *and* negative cases (i.e. the hypothesis should not be awarded for its inability to distinguish between positive and negative samples). This error was unlikely to be triggered in the framework used by Sebag et al., but it is not uncommon to meet constant functions in the first generations of a GP run.

As we are maximizing the AUC, it is quite natural to consider the area *above* the curve as a measure of the error (or loss) of the hypothesis. If one examines the weighted AUC fitness function, one notices that it can also be used to draw the ROC curve: starting from the origin (0% true positive, 0% false positive) and reading every case in the sort order, every positive case $(x_i, 1)$ draws a vertical segment of height w_i and every negative case $(x_j, -1)$ draws a horizontal segment of width w_j. Here we consider the area of the rectangle *above* a negative case as its loss (see illustration in Fig. 2), so it amounts to $w_j * \Sigma_{(x_i,1),j+1 \leq i \leq n} w_i$ (using the sort order and not the initial random order of samples). In the same way, the loss of a positive case is the area of the rectangle located to the *left* of its vertical segment: this amounts to $w_i * \Sigma_{(x_j,-1),1 \leq j \leq i-1} w_j$. The update procedure is given in Table 3.

When doing the above computation, note that the area above the ROC curve would be summed up twice, thus we correct this by dividing each individual loss by 2. Then it is normalized so that the whole ROC diagram area sums up to 1. The global cumulative loss is always between 0 (hypothesis can achieve perfect classification by choosing appropriate threshold) and 1 (there is a threshold such that every sample is misclassified). This gives us a sensible individual loss to affect the weight of each case, that is closely related to the fitness value of the hypothesis.

[1] http://www.ics.uci.edu/~mlearn/MLRepository.html

Table 2. Computation of the weighted area under the ROC curve (see also [10])

Function weightedAUC

```
Input
   Data set S = {(x_i, y_i)}, i = 1...n, x_i ∈ X, y_i ∈ {1, −1}
   Hypothesis h : X → ℝ // e.g. a GP tree in our experiments
   Weights set W = {w_i}, i = 1...n, w_i ∈ [0, 1], Σ_1^n w_i = 1
Begin
   Sort S = {(x_i, y_i)} by decreasing order, where i > j
        iff (h(x_i) > h(x_j)) or ((h(x_i) = h(x_j) and (y_i < y_j)).
   p=0
   F = 0
   for i = 1 to n
      if y_i = 1 then p=p+w_i
      else F =F+p*w_i
   end for
   F = F/(Σ_(x_i,1) w_i * Σ_(x_j,−1) w_j) // normalize F
   return F
End
```

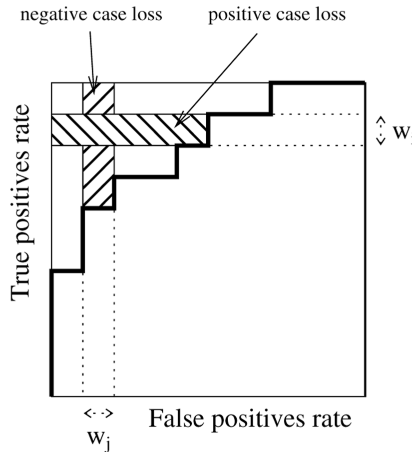


Fig. 2. Illustration of the loss area of a positive and negative cases for the weights update algorithm

The resulting update formula uses the global and individual losses to compute the confidence value of the hypothesis and to update the weights for the next boosting round. This is done in the same way as for standard boosting regression problems see e.g. [16]. The final combined predictor is obtained by sorting the hypotheses outputs and keeping the one associated to the median of the cumulative confidences, see Table 4 and also [16,17].

Table 3. Updating the weights in relation with the area *above* the ROC curve

```
Function updateWeights
Input
   Data set S = {(x_i, y_i)}, i = 1...n, x_i ∈ X, y_i ∈ {1, −1}
   Weights set W = {w_i}, i = 1...n, s.t. Σ_{i=1}^n w_i = 1
   Hypothesis h : X → ℝ // e.g. a GP tree in our experiments
Begin
   loss = 0; // global loss, i.e. area above curve
   lossSamples[] = 0; // array of loss for each sample
   Sort S = {(x_i, y_i)} by decreasing order, where i > j
        iff (h(x_i) > h(x_j)) or ((h(x_i) = h(x_j) and (y_i < y_j)).
   // compute area left of positive cases or above negative cases
   for i = 1 to n
      if y_i = 1 then // positive case
         lossSamples[i] = w_i * Σ_{(x_j,−1),1≤j≤i−1} w_j
      else // negative case
         lossSamples[i] = w_i * Σ_{(x_j,1),i+1≤j≤n} w_j
      end if
      lossSamples[i] = lossSamples[i]/2 // area should not be summed twice
      // normalize so that ROC diagram area = 1
      lossSamples[i] = lossSamples[i]/(Σ_{(x_i,1)} w_i * Σ_{(x_j,−1)} w_j)
      loss = loss + lossSamples[i]
   end for
   // Update weights
   β =loss /(1−loss)
   for i = 1 to n
      w_i = w_i * β^(1.0−lossSamples[i])
   end for
   normalize W so that Σ_{i=1}^n w_i = 1
   return log(1/β) // this is the confidence
End
```

3 Experiments with ROC-Based GP and ES

3.1 Experimental Settings

To validate our ROCboost algorithm, we performed a set of experiments following the general framework set in [10] that seems a suitable comparison point for evolutionary ROC-based learning. We used 8 benchmarks from the University of California – Irvine repository, namely the `breast-cancer`, `german`, `crx`, `votes`, `promoters`, `satimage` discriminating between classes 3 and 4, `vehicles` for classes `saab` and `bus` and `waveform` for classes 1 and 2. Every data set has been processed as in [10]: we take 11 random splits between learning and test data, we keep the same proportion of positive and negative cases in learning and test sets, and we make 21 independent runs on each split, thus yielding 231

Table 4. Pseudo code for ROCboost algorithm

```
let S = {(x_i, y_i)}, i = 1,...,n, x_i ∈ X, y_i ∈ {1,−1}, be the training samples set.
initialize the weights w_i = 1/n, i = 1,...,n
for r in 1..t // iterate t boosting rounds
   learn hypothesis h_r(x) → ℝ using training samples weighted by w_i
                              and fitness function weightedAUC
   compute confidence c_r and update weights using function updateWeights
end for
let (x,?) be a test sample.
output combined predictor: inf { y ∈ {h_r(x)}_{r∈1...n} : Σ_{t:h_t(x)≤y} c_t ≥ 1/2 Σ_{r∈1...n} c_r }
```

Table 5. ROCboost parameters for GP

Function set	+, -, *
Terminal set	ERC in $[-2.0, +2.0]$, {ProblemInputs}
Population size	200
♯ generations	51
Crossover rate	.9
Mutation rate	.05
Copy rate	.05
Elitism (♯ individuals)	2
Tournament selection	5
Max tree depth	17
♯ boosting rounds	100

independent runs on each data set in order to gather some statistical consistency since we are using stochastic learning. However, we depart from [10] since we consider both non linear GP trees and linear functions optimized with ES. The number of boosting rounds was set to 100.

We used the ECJ[2] library for all experiments. The parameters for the GP runs are shown in Table 5 while those for the ES are taken from [10].

For GP we used 3 operators for internal nodes $(+,-,*)$, ephemeral random constants (ERC) in the range $[-2.0, +2.0]$, and as many inputs as needed by the problem at hand. Note that we performed what could be called as "lazy" data preparation for GP: qualitative features were simply given successive integer numbers rather than being partitioned in separate boolean features as is common practice. We chose to do so because we wanted to test the ability of GP to work with such unrefined data, and also because we felt that medium size or large sets of terminals implied by separation in boolean features could maybe hamper the GP process. This last point would need further study in itself.

[2] http://cs.gmu.edu/~eclab/projects/ecj/

Table 6. The AUC values on the test sets of plain GP, ROCboost GP, ES, ROCboost ES, ROGER and Support Vector Machines (last two issued from [10]) on eight data sets from the Irvine Repository (names are abbreviated)

	plain GP	ROCboost GP	plain ES	ROCboost ES	ROGER	SVMTorch
Breast	.647 ± .044	.664 ± .042	.642 ± .052	.650 ± .051	**.674 ± .05**	.672 ± .05
Crx	.862 ± .022	**.899 ± .022**	.805 ± .028	.839 ± .018	.816 ± .06	.839 ± .04
German	.709 ± .013	**.746 ± .014**	.725 ± .024	.745 ± .023	.712 ± .03	.690 ± .02
Promot.	.719 ± .026	.849 ± .06	.797 ± .032	.968 ± .017	.863 ± .07	**.974 ± .02**
SatImg	**.921 ± .009**	.917 ± .007	.911 ± .013	.918 ± .012	.918 ± .01	.876 ± .02
Vehicle	.950 ± .007	.972 ± .007	.946 ± .009	.961 ± .007	**.994 ± .005**	.993 ± .007
Votes	.984 ± .005	.990 ± .005	.983 ± .005	.990 ± .003	**.993 ± .004**	.989 ± .005
Wave	.983 ± .002	**.988 ± .002**	.986 ± .002	**.988 ± .002**	.971 ± .004	.963 ± .008

Table 7. Wilcoxon test assessing the improvement brought by ROCboost for GP and ES on the test sets

	♯ att GP	♯ att ES	Nb samples ♯Train	Nb samples ♯Test	p-value: boosted GP > GP?	p-value: boosted ES > ES?
Breast Cancer	9	42	189	97	0.995	0.999
Crx	15	47	70	620	0.999	0.999
German	24	24	100	900	0.999	0.999
Promoters	57	228	70	36	0.999	0.999
SatImage 3-4	36	36	139	1237	*.021*	0.999
Vehicle saab-bus	18	18	125	291	0.999	0.999
Votes	16	32	287	148	0.999	0.998
Waveform 1-2	21	21	211	3131	0.999	0.999

When we perform a boosting experiment, the first round hypothesis is indeed a plain GP (or ES) run. We have gathered plain GP (respectively ES) and boosted GP (resp. ES) results on the test set, and we took the median AUC values from each set of 21 experiments on a given split, then averaging these medians over the 11 splits for a given problem, to obtain figures comparable to [10].

3.2 Results

The main point of inquiry was whether boosted GP (resp. ES) improves on plain GP (resp. ES). Results are summed up in Table 6 and the significance is checked with a paired Wilcoxon statistical test, whose results are shown in Table 7. We can observe that it is true with a significant probability, except for one case, GP on `satimage`.

This behavior is illustrated for GP by average learning curves on the training and test sets, plotting average AUC versus boosting rounds in Fig. 3. Note that since we plot the learning curve of the combined predictor, it can decrease on the training set, even when using elitism for the individual hypothesis. In all but one cases, 100 rounds of boosting do not over-fit GP nor ES, even with maximum

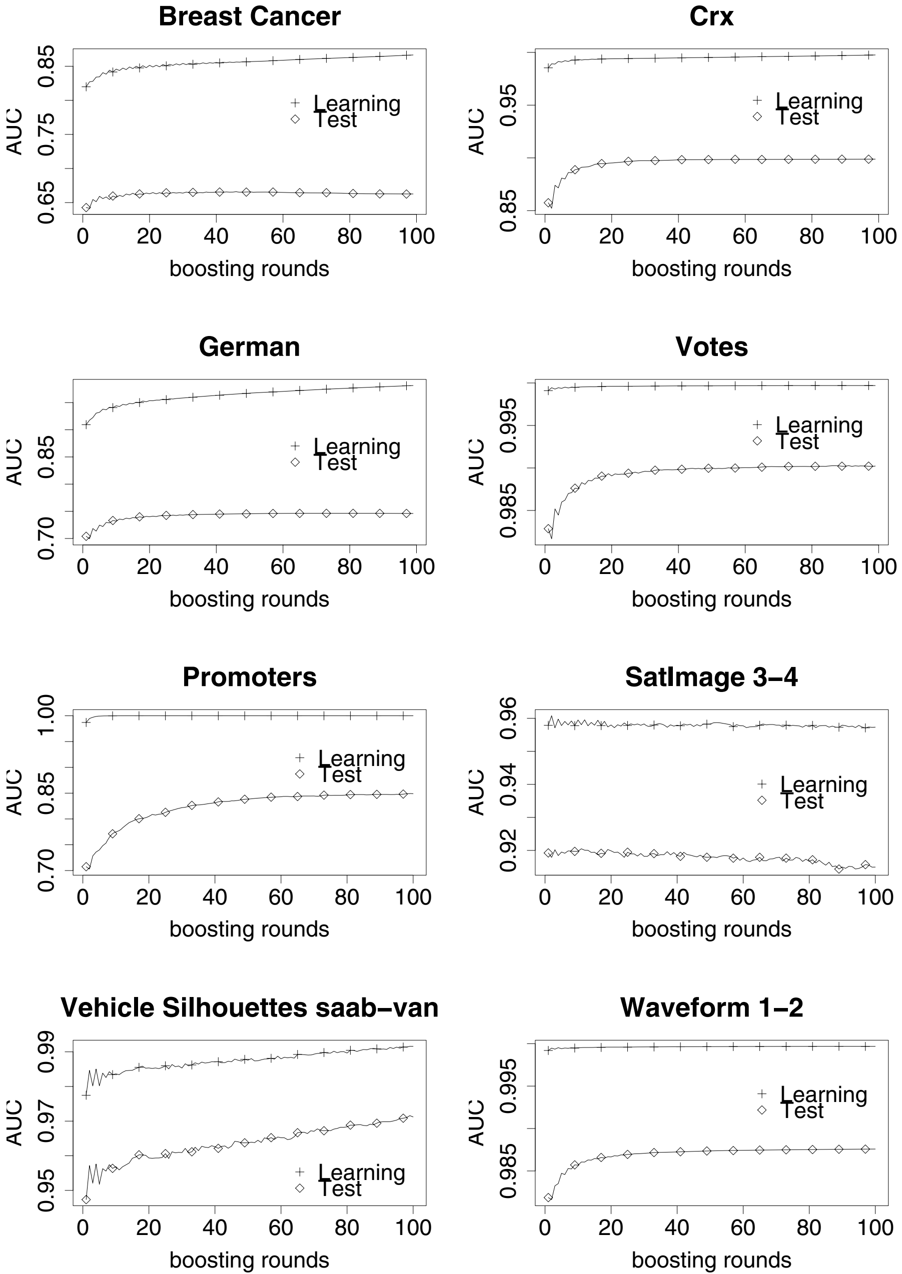


Fig. 3. ROCboost GP: average AUC of learning and test samples versus boosting rounds

depth 17 trees for GP (i.e. with the possibility to evolve complex hypotheses). Even for the `satimage` benchmark, the over-fitted boosted GP hypothesis remains competitive with the other learners.

When it comes to comparison between our results and those from ROGER or SVMTorch, we urge the reader to take some caution, because:

- GP and boosted GP do not work in the same hypothesis space than ROGER and SVMTorch (and it would seem artificial to design a GP version that explores only linear functions);
- even if standard deviations are reported in [10], most often the distribution of results is not normal according to a Shapiro-Wilk test, thus comparisons are to be taken as rough indicators; this is emphasized by the "plain ES" column that ought to yield results similar to ROGER since we used a setting as close as we could.

Nonetheless we can observe that the performance of plain GP, with the same number of evaluations than ROGER and ES, is somewhat contrasted: it gives the worst results on the `promoter` data set, but also the best on `satimage`. It also improves much over ES, ROGER and SVM for the `crx` benchmark, suggesting that linear hypotheses are not adequate for this data set.

We can also conclude that boosting GP and ES give competitive performance, obviously at a much increased cost in computing time, since each boosting round cost is equivalent to a standard run of the learner. Typical running times for 100 boosting rounds on these experiments ranged from 10 to 30 minutes on a 1,5Ghz laptop PC.

4 Conclusion

From the previous section results, we see that ROCboost generally improves GP and ES up to the point of being competitive in terms of performance with the other heuristics. It avoided to over-fit on all benchmarks but GP on `satimage`; even in this case the resulting performance was still comparable to the other learners. This last point is important, and this is why we purposely worked with maximum depth 17 trees in order to test complex hypotheses that might have triggered over-fitting.

We experimented the ROCboost scheme with standard GP and ES and observed similar results, thus showing it allows to perform AUC optimization with standard evolutionary learners that does not respect the constraints required by Freund et al.'s RankBoost algorithm. A drawback is the increased computing time, as with every boosting scheme, since we iterate the learning process.

Although our boosting scheme has shown effective improvements in terms of AUC optimization on several benchmarks, the paper title emphasizes the empirical nature of these results since we have no formal proof of the reduction of the error. This is obviously a goal to achieve in future works.

References

1. Schapire, R.E.: The strength of weak learnability. Machine Learning **5**(2) (1990) 197–227
2. Freund, Y.: Boosting a weak learning algorithm by majority. Information and Computation **121**(2) (1995) 256–285
3. Freund, Y., Schapire, R.E.: Experiments with a new boosting algorithm. In: Machine Learning: Proceedings of the Thirteenth International Conference, Morgan Kaufmann (1996) 148–156
4. Breiman, L.: Bagging predictors. Machine Learning **24** (1996) 123–140
5. Friedman, J., Hastie, T., Tibshirani, R.: Additive logistic regression: a statistical view of boosting. The Annals of Statistics **28**(2) (2000) 237–407
6. Bradley, A.: The use of the area under the roc curve in the evaluation of machine learning algorithms. Pattern Recognition **30**(7) (1997) 1145–1159
7. Provost, F.J., Fawcett, T., Kohavi, R.: The case against accuracy estimation for comparing induction algorithms. In: Proceedings of the Fifteenth International Conference on Machine Learning, Morgan Kaufmann (1998) 445–453
8. Cohen, W.W., Schapire, R.E., Singer, Y.: Learning to order things. In: NIPS '97: Proceedings of the 1997 conference on Advances in neural information processing systems 10, Cambridge, MA, USA, MIT Press (1998) 451–457
9. Mozer, M.C., Dodier, R., Colagrosso, M., Guerra-Salcedo, C., Wolniewicz, R.: Prodding the roc curve: constrained optimization of classifier performance. In: Advances in Neural Information Processing Systems. Volume 14. The MIT Press (2001)
10. Sebag, M., Azé, J., Lucas, N.: ROC-based evolutionary learning: Application to medical data mining. In: Proc of the 6th Artificial Evolution conference EA'03. Volume 2936 of LNCS., Springer (2004) 384–396
11. Sebag, M., Azé, J., Lucas, N.: Impact studies and sensitivity analysis in medical data mining with ROC-based genetic learning. In: Data Mining, 2003. ICDM 2003. Third IEEE International Conference on. (2003) 637–640
12. Azé, J., Lucas, N., Sebag, M.: A genetic roc-based classifier. Technical report, LRI, Orsay, France (July 2004)
13. Freund, Y., Iyer, R., Schapire, R.E., Singer, Y.: An efficient boosting algorithm for combining preferences. Journal of Machine Learning Research **4**(6) (2003) 933–969
14. Cortes, C., Mohri, M.: Auc optimization vs. error rate minimization. In Thrun, S., Saul, L., Schölkopf, B., eds.: Advances in Neural Information Processing Systems 16. MIT Press, Cambridge, MA (2004)
15. Iba, H.: Bagging, boosting, and bloating in genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference GECCO'99, Morgan-Kaufmann (1999) 1053–1060
16. Drucker, H.: Improving regressors using boosting techniques. In: Proceedings of the 14th International Conference on Machine Learning (ICML), Morgan Kaufmann (1997) 107–115
17. Paris, G., Robilliard, D., Fonlupt, C.: Applying boosting techniques to genetic programming. In: Proceedings of the 5th Artificial Evolution conference EA'01. Volume 2310 of LNCS., Springer (2002) 245–254

Confidence Intervals for Computational Effort Comparisons

Matthew Walker, Howard Edwards, and Chris Messom

Institute of Information and Mathematical Sciences,
Massey University, Auckland, New Zealand
{m.g.walker,h.edwards,c.h.messom}@massey.ac.nz

Abstract. When researchers make alterations to the genetic programming algorithm they almost invariably wish to measure the change in performance of the evolutionary system. No one specific measure is standard, but Koza's computational effort statistic is frequently used [8]. In this paper the use of Koza's statistic is discussed and a study is made of three methods that produce confidence intervals for the statistic. It is found that an approximate 95% confidence interval can be easily produced.

1 Introduction

In *Genetic Programming* [6], Koza described a statistic to assess the computational burden of using GP. It calculates the minimum number of individuals that must be evaluated in order to yield a solution 99% of the time. This statistic, minimum computational effort, E, was used heavily throughout Koza's first two books on GP [6,7] to compare the performance of variations of GP.

Given the cumulative probability of success, $P(i)$, of a number of GP runs, we can calculate how many runs would be required, $R(i, z)$, in order to find a solution at generation i with probability z[1]:

$$R(i, z) = \left\lceil \frac{\log(1 - z)}{\log(1 - P(i))} \right\rceil \tag{1}$$

The computational effort, $I(i, z)$ (the number of individuals that need to be evaluated to find a solution with probability z) for generation i with a population of M individuals is calculated by:

$$I(i, z) = i \times R(i, z) \times M \tag{2}$$

Koza's minimum computational effort, E, is the minimum value of $I(i, z)$ over the range of generations from 0 to the maximum in the experiment.

Christensen and Oppacher [3] recommended the ceiling operator ($\lceil \cdot \rceil$) should be dropped and indeed Koza himself had done just that up to seven years earlier [1].

[1] As is common, z will be set to 0.99 throughout this work.

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 23–32, 2007.
© Springer-Verlag Berlin Heidelberg 2007

We suggest that the number of runs required should have a lower bound of one. This means that if the ceiling operator is to be dropped, then

$$R(i,z) = \begin{cases} \frac{\log(1-z)}{\log(1-P(i))} \text{ if } P(i) < z \\ 1 \text{ if } P(i) \geq z \end{cases} \quad (3)$$

Although the use of the ceiling operator tends to overestimate the computational burden, Christensen and Oppacher [3] showed that the use of the minimum operator tends to produce an underestimate of the true computational effort required.

A number of researchers have commented that, for experiments where cumulative probability of success is low, a small change in the measurement of the number of successful experiments can have a significant impact on the computational effort calculation [9,8,11]. Thus a lower confidence level should be associated with computational effort statistics based on low success rates. Unfortunately, some quoted minimum computational effort statistics do not state the number of runs that were successful.

Niehaus and Banzhaf [11] demonstrated that doubling the number of runs can more than halve the range of observed computational effort values. But as Angeline [2] pointed out, a key problem with Koza's computational effort statistic is that, as defined, it is a point statistic with no confidence interval. Without a confidence interval, comparisons are inconclusive.

2 Defining Confidence Intervals

2.1 Normal Approximation Method

This section discusses how an approximate 95% confidence interval can be generated for a computational effort statistic if the true *minimum generation* is known. The minimum generation is the generation at which the minimum computational effort occurs. The method used in this section is the textbook "normal approximation" method.

The cumulative probability of success statistic is calculated from the proportion of the population that has found a solution at a given generation. We may assume that this proportion is approximately normally distributed [4] and thus calculate an approximate 95% confidence interval using $p \pm e$ where $e = 2\sqrt{\frac{p(1-p)}{n}}$, p is the proportion of runs that found a solution by the specified generation and n is the number of runs performed. Given that cumulative success cannot be below zero or above one, the confidence interval should be truncated to that range [10].

The minimum and maximum of this confidence interval can be used to generate an approximate 95% confidence interval for $\mathcal{R}$, the true number of runs required to find a solution with probability z:

$$\frac{\log(1-z)}{\log(1-(p+e))} \leq \mathcal{R} \leq \frac{\log(1-z)}{\log(1-(p-e))} \quad (4)$$

If the minimum and maximum of this range is used in place of $R(i, z)$ in the formula for computational effort (equation 2), the values can be used as an approximate 95% confidence interval for the true value of Koza's computational effort statistic, $\mathcal{I}(i)$, for generation i:

$$i \times M \times \frac{\log(1-z)}{\log(1-(p+e))} \leq \mathcal{I}(i) \leq i \times M \times \frac{\log(1-z)}{\log(1-(p-e))} \quad (5)$$

These confidence intervals are only valid while $np > 5$ and $n(1-p) > 5$ [4] where $p = P(i)$ and n is the number of runs that were executed.

2.2 Wilson's Method

This section discusses the replacement of the normal approximation method with Wilson's 'score' method [10] when constructing confidence intervals for the computational effort statistic. It is still assumed that the minimum generation is known.

To calculate a 95% confidence interval for the true but unknown proportion of successes based on the observed sample proportion of successes, $p = r/n$, given r successes from n runs, these formulae [10] may be used (where the standard normal variable $z_{\text{norm}} = 1.96$):

$$\mathit{upper}(p, n) = \frac{2np + {z_{\text{norm}}}^2 + z_{\text{norm}}\sqrt{{z_{\text{norm}}}^2 + 4np(1-p)}}{2(n + {z_{\text{norm}}}^2)} \quad (6)$$

$$\mathit{lower}(p, n) = \frac{2np + {z_{\text{norm}}}^2 - z_{\text{norm}}\sqrt{{z_{\text{norm}}}^2 + 4np(1-p)}}{2(n + {z_{\text{norm}}}^2)} \quad (7)$$

Robert Newcombe compared seven statistical methods for producing confidence intervals for a proportion [10]. He demonstrated the normal approximation method suffers from overshoot (where the confidence interval goes below zero or above one), the possibility of producing zero-width intervals, and that it has an estimated mean *coverage* of just 0.88 (compared to the expected 0.95). Coverage is the term for the percentage of confidence intervals that included the best estimate of the true value of computational effort. He also showed that Wilson's method suffers from neither overshoot nor the possibility of producing zero-width intervals, and it has an estimated mean coverage of 0.952.

Using the formulae in equations 6 and 7, a confidence interval can be established for the proportion of successful runs: the upper bound is given by $\mathit{upper}(P(i), n)$ and the lower bound is given by $\mathit{lower}(P(i), n)$. Just as was done in the previous section, the minimum and maximum of this range can then be used to calculate a maximum and minimum for the number of runs required to obtain a solution with probability z. These numbers can then be used with the known value for the population size, M, to find a 95% confidence interval for the true computational effort, $\mathcal{I}(i)$, at a given generation i:

$$i \times M \times \frac{\log(1-z)}{\log(1-\mathit{upper}(p, n))} \leq \mathcal{I}(i) \leq i \times M \times \frac{\log(1-z)}{\log(1-\mathit{lower}(p, n))} \quad (8)$$

When the minimum generation is known, Wilson's method produces a valid confidence interval irrespective of the number of runs or the probability of success.

2.3 Resampling Statistics Method

Keijzer et al. [5] are the only group we have found who attempted to generate a confidence interval for Koza's computational effort statistic. We implemented a bastardised version of their method (see table 1). When the true minimum generation is known, the minimum computational effort is calculated for the selection as the selection's computational effort at the true minimum generation. The resampling method always finds a confidence interval irrespective of the number of runs and the probability of success.

Table 1. Algorithm for the Resampling method

1. Obtain n independent runs. Label these as the source set.
2. Repeat 10,000 times:
   (a) Select, with replacement, n runs from the source set.
   (b) Calculate the minimum computational effort statistic for the selection. If zero runs succeeded, the computational effort is infinite.
3. Find the 2.5% and 97.5% quantiles of the 10,000 computational effort statistics. These provide an upper and lower range on a 95% confidence interval for the true minimum computational effort.

3 When Minimum Generation Is Known

3.1 Testing the Validity of the Three Methods

In order to empirically test the validity of these three methods to generate confidence intervals, we ran experiments based on datasets where very large numbers of runs had been executed[2]. The four datasets were:

- *Ant*: Christensen and Oppacher's 27,755 runs [3] of the artificial ant on the Santa-Fe trail; panmictic population of 500; best estimate of the true computational effort 479,344 at generation 18[3]; $P(18) = \frac{2421}{27755} = 0.0872$
- *Parity*: 3,400 runs of even-4-parity without ADFs [6,7]; panmictic population of 16,000; best estimate of true computational effort 421,074 at generation 23; $P(23) = \frac{3349}{3400} = 0.985$

[2] The datasets and complete results are available for download from www.massey.ac.nz/~mgwalker/CompEffort.

[3] This occurred at generation 18 as, like Koza, we have counted the first generation as generation 0, whereas Christensen and Oppacher labelled it generation 1.

- *Symbreg*: Gagné's 1,000 runs[4] of a symbolic regression problem ($x^4 + x^3 + x^2 + x$) [6]; panmictic population of 500; best estimate of true computational effort 33,299 at generation 12; $P(12) = \frac{593}{1000} = 0.593$
- *Multiplexor*: Gagné's 1,000 runs[5] of the 11-multiplexor problem [6]; panmictic population of 4,000; best estimate of true computational effort 163,045 at generation 25; $P(25) = \frac{947}{1000} = 0.947$

The computational effort calculations for each dataset (utilising every run) were treated as a best estimate of the true minimum generation and true minimum computational effort.

For each dataset and for each confidence interval generating method, the following method was applied. A subset of the whole dataset's runs were randomly selected (uniformly with replacement). The subset sizes were 25, 50, 75, 100, 200 and 500 runs. These sizes are typical of published work (often 25 to 100 runs, sometimes fewer [6,7]) and recommendations by statisticians (200 to 500 runs [3,11]). 10,000 subsets were selected and for each subset the confidence interval generating method was applied. This simulated 10,000 genetic programming experiments on each of the four problem domains for each of the six run sizes.

3.2 Results and Discussion

For each of the four problem domains and each of the three confidence interval generation methods, table 2 gives the average coverage and the average number of confidence intervals that were produced from the 10,000 simulated experiments. Table 3 gives the same statistics but by run size and method.

So, for example, table 2 shows that for the normal approximation method on the Ant problem domain, an average of 97.1% of the confidence intervals included the true value of the minimum computational effort (compare that to the expected result of approximately 95%). This average was produced over simulated experiment sizes of 25–500 runs. The table also shows that, for the same setup, an average of 7,049 of the 10,000 simulated experiments produced valid confidence intervals.

The resampling method had a very poor minimum average coverage of 69.9% for the Parity domain (see table 2). The Normal method also did poorly for that domain with a coverage score of 49.4%. In contrast, the Wilson method achieved very good coverage levels across all domains and all run sizes with a minimum coverage of 93.3% (on the Parity domain with 100 runs).

The advantage of Wilson's method over the normal approximation method is clearly demonstrated by the validity statistics in the Parity problem. Because the probability of success is so high (0.985 over 3,400 runs), the samples with a low number of runs (25–200) were often unable to satisfy the normal method's validity criteria of $n(1-p) > 5$. And even when the validity criteria were satisfied, for the small runs sizes (i.e. 50 and 75 runs), none of the confidence intervals

[4] Our thanks go to Christian Gagné for this dataset.

[5] Thanks again to Christian Gagné for this dataset.

Table 2. Average coverage percentages and average validity statistics by problem domain when the minimum generation is known. Averages are over 25–500 runs.

Method \ Problem	Ant	Parity	Symbreg	Multiplexor	Average
Normal	97.1%	49.4%	95.3%	79.1%	80.3%
	7,049	1,787	9,954	4,752	5,885
Wilson	95.2%	95.3%	94.9%	95.1%	95.1%
	10,000	10,000	10,000	10,000	10,000
Resampling	93.2%	69.9%	94.1%	88.9%	86.5%
	10,000	10,000	10,000	10,000	10,000

Table 3. Average coverage percentages and average validity statistics by run size when the minimum generation is known. Averages are over the four problem domains.

Method \ Runs	25	50	75	100	200	500	Average
Normal	48.1%	70.5%	72.9%	98.0%	96.3%	95.9%	80.3%
	2,582	3,704	5,007	6,439	7,907	9,674	5,885
Wilson	94.6%	95.7%	95.6%	94.7%	95.5%	94.7%	95.1%
	10,000	10,000	10,000	10,000	10,000	10,000	10,000
Resampling	72.0%	82.8%	86.9%	88.8%	94.4%	94.3%	86.5%
	10,000	10,000	10,000	10,000	10,000	10,000	10,000

included the best estimate of the true computational effort. Wilson's method, on the other hand, produced valid confidence intervals for all 10,000 samples for every run size and with a coverage of 95.3% for the experiments in that domain. Where it was fair to make a comparison, the widths of the confidence intervals were similar.

The Ant domain exemplifies a low probability of success ($P(18) = 0.087$). In this case the Normal method had difficulty satisfying its $np > 5$ criteria, producing valid confidence intervals for only 6% of the samples with 25 runs and 43% with 50 runs. However, for the confidence intervals that it did produce, the proportions that included the true value either exceeded or were very close to the intended 95%. However, yet again the Wilson method was the method of choice as it produced confidence intervals for every sample and with an average coverage of 95.2%. Further, for almost every run size Wilson's method produced notably tighter confidence intervals.

Finally, the Symbreg domain, with its non-extreme cumulative probability of success ($P(12) = 0.593$), levelled the playing field for the Normal method. The Normal method produced very good average coverage of 95.3% for an average of 99.5% of the samples. The Wilson method did only slightly better in this instance, although the widths of its confidence intervals were a little tighter.

The Resampling method did very poorly over lower (25–100) run counts for the parity problem (coverages of 32%–78%). This was due to the low probability that a sample of the population would contain a run that did *not* find a solution before the minimum generation. For data where the cumulative success rate is

very high at the minimum generation, it can now be seen that the resampling method is inappropriate to use.

4 When the Minimum Generation Is Unknown

4.1 Changes to the Methods

Unfortunately, is it extremely unlikely that a researcher will know the number of generations at which the true minimum computational effort occurs. This section discusses how confidence intervals can be established using an estimate of the true minimum generation.

For a sample of genetic programming runs, the minimum generation can be estimated by using the technique Koza described. That is, by calculating the computational effort, $I(i)$, for every generation, i, from 0 to the maximum in the experiment. The estimated minimum generation is the generation where $I(i)$ is minimal.

For the generation of confidence intervals, the estimated minimum generation is used in place of the true minimum generation, but otherwise the three methods remain unchanged.

From a statistical perspective this introduces dependence between the measurements of minimum generation and the minimum computational effort. Keijzer et al. suggested that the runs in a GP experiment could be divided into two halves; the first half used to estimate the minimum generation and the second half used to estimate the minimum computational effort. However the cost of a GP run is typically so expensive that using only half the runs to establish computational effort is not seriously considered. This work follows that pragmatic approach and accepts the dependence.

Because no effort has been made to account for the increased variability in the estimated computational effort that is due to estimating the minimum generation, it should be expected that the confidence intervals produced using these methods would achieve less than 95% coverage.

4.2 Results and Discussion

For each problem domain and confidence interval generation method, table 4 gives the average coverage and the average number of valid confidence intervals that were produced. Table 5 gives the same statistics but by run size and method.

Figure 1 depicts box and whisker plots of the width of the confidence intervals produced using each of the three methods for each of the six run sizes on the Ant domain. The grey line across each plot indicates the value of the best estimate of the true computational effort. This line is added to assist understanding of the magnitude of the widths. The whiskers (indicated by the dashed line) in the plots extend to the most extreme data point or 1.5 times the interquartile range from the box, whichever is smaller. In the latter case, points past the whiskers are considered outliers and are marked with a small circle. The box-plot for

Table 4. Average coverage percentages and average validity statistics by problem domain when the minimum generation is estimated. Averages are over 25–500 runs.

Method \ Problem	Ant	Parity	Symbreg	Multiplexor	Average
Normal	96.1%	63.8%	94.8%	93.1%	86.9%
	7,012	1,892	9,839	3,684	5,606
Wilson	92.9%	94.0%	94.9%	95.7%	94.4%
	9,950	10,000	10,000	10,000	9,988
Resampling	92.4%	65.3%	91.2%	72.3%	80.3%
	10,000	10,000	10,000	10,000	10,000

Table 5. Average coverage percentages and average validity statistics by run size when the minimum generation is estimated. Averages are over the four problem domains.

Method \ Runs	25	50	75	100	200	500	Average
Normal	65.1%	72.3%	94.7%	97.3%	96.7%	95.4%	86.9%
	2,497	3,770	4,695	5,595	7,212	9,870	5,606
Wilson	93.0%	94.4%	94.7%	93.8%	94.9%	95.3%	94.4%
	9,928	9,998	10,000	10,000	10,000	10,000	9,988
Resampling	62.2%	73.9%	80.2%	84.5%	89.0%	91.9%	80.3%
	10,000	10,000	10,000	10,000	10,000	10,000	10,000

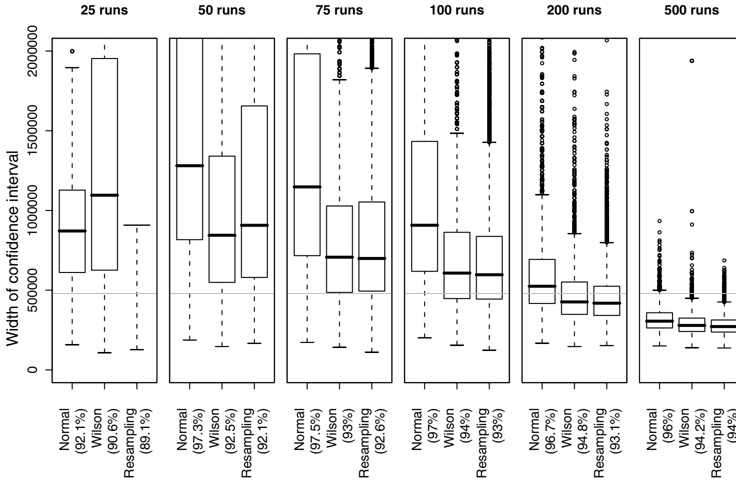


Fig. 1. Confidence interval widths for the Ant problem domain when the minimum generation is estimated. Percentages indicate coverage for the specific configurations.

25 runs using the Resampling method is incomplete as more than 50% of the simulated experiments produced infinite confidence interval widths.

Surprisingly, the use of an estimated minimum generation had very little negative impact on the coverage of the three methods. Excluding the Parity domain,

the Normal method did well with an average coverage of 94.7% (as against the intended 95%). Wilson's method did even better as as, over *all* problem domains and run sizes, it dropped only slightly to an average of 94.4% (as compared to 95.2% when the true minimum generation was known). From these results it appears that, even when an estimated minimum generation is used, the confidence intervals produced by the Wilson method are a good approximation to a 95% confidence interval.

It is hypothesised that the use of an estimated minimum generation had so little negative effect because the computational effort performance curves flatten out around the true minimum generation, and that the use of an estimate provides a result "good enough" for the production of a confidence interval.

Finally, it is very significant that the median widths of the confidence intervals are almost always greater than the best estimate of the true value.

5 Further Analysis of Wilson's Method

Although it is easy to retrospectively apply Wilson's method to previously published results, because it is common for published work to fail to give the cumulative success proportion or the minimum generation at which the computational effort was calculated, Wilson's method isn't always able to be applied. We can instead consider a "best-case" confidence interval, one that gives the smallest range of computational effort given the number of runs executed. In this way we can say that a 95% confidence interval is *at least* this range.

To calculate the "best-case" for the lower bound, take the minimum defined value for each run size across the range of possible values for cumulative probability of success. More formally,

$$\min_{p} \frac{R(upper(p,n)) - R(p)}{R(p)} \tag{9}$$

where p ranges from 0 to 1; *upper* is the upper bound of a 95% confidence interval of a proportion (see equation 6); n is the number of runs; and

$$R(i,z) = \begin{cases} \frac{\log(1-z)}{\log(1-P(i))} \text{ if } P(i) < z \\ \textit{undefined} \text{ if } P(i) \geq z \end{cases} \tag{10}$$

The "best-case" for the upper bound is calculated in a similar way, but with *upper* replaced with *lower*, the lower bound in equation 7. The "best-case" scenario is only valid if $upper(P(i), n)) < z$.

This approach can be used if a computational effort value, E, has been stated for a specified number of runs, say 50, but without a value for the cumulative probability of success. In this case we can use the above formulae to calculate a "best-case" confidence interval for $\mathcal{E}$, the true minimum computation effort, as $(1 - 0.26)E \leq \mathcal{E} \leq (1 + 0.45)E$. The true 95% confidence interval will be *at least* this size.

6 Conclusions

Wilson's method is an appropriate way to produce confidence intervals for Koza's computational effort statistic. From the empirical results, the use of an estimated minimum generation has little effect on the coverage and the intervals can be treated as a very good approximation to 95% confidence intervals.

The Wilson method can often be retrospectively applied to earlier work as only the number of runs and the success proportion at the generation of minimum computational effort are required. If the number of runs is known but the success proportion is not known, then a minimum confidence interval can be generated using the "best-case" approach. Applying these confidence intervals may cast doubt on the validity of some published results.

Finally, computational effort may not be the best measurement for comparison as this study has shown that results that differ by 50% or 100% may, from a statistical perspective, not be significantly different.

References

1. David Andre and John R. Koza. Parallel genetic programming on a network of transputers. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming*, pages 111–120, 1995.
2. Peter J. Angeline. An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, 1996.
3. Steffen Christensen and Franz Oppacher. An analysis of Koza's computational effort statistic for genetic programming. In *Genetic Programming, Proceedings of the 5th European Conference, EuroGP*, volume 2278 of *LNCS*. Springer-Verlag, 2002.
4. G. Clarke and D. Cooke. *A Basic Course in Statistics*. Arnold, 4th edition, 1998.
5. M. Keijzer, V. Babovic, C. Ryan, M. O'Neill, and M. Cattolico. Adaptive logic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 42–49. Morgan Kaufmann, 2001.
6. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.
7. John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, 1994.
8. Sean Luke and Liviu Panait. Is the perfect the enemy of the good? In *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann, 2002.
9. J. Miller and P. Thomson. Cartesian genetic programming. In *Genetic Programming, Proceedings of EuroGP*, volume 1802 of *LNCS*, 2000.
10. Robert G. Newcombe. Two-sided confidence intervals for the single proportion: comparison of seven methods. *Statistics in Medicine*, 17:857–872, 1998.
11. Jens Niehaus and Wolfgang Banzhaf. More on computational effort statistics for genetic programming. In *Genetic Programming, Proceedings of EuroGP*, volume 2610 of *LNCS*. Springer-Verlag, 2003.

Crossover Bias in Genetic Programming

Maarten Keijzer[1] and James Foster[2]

[1] mkeijzer@xs4all.nl
[2] University of Idaho
foster@uidaho.edu

Abstract. Path length, or search complexity, is an understudied property of trees in genetic programming. Unlike size and depth measures, path length directly measures the balancedness or skewedness of a tree. Here a close relative to path length, called visitation length, is studied. It is shown that a population undergoing standard crossover will introduce a crossover bias in the visitation length. This bias is due to inserting variable length subtrees at various levels of the tree. The crossover bias takes the form of a covariance between the sizes and levels in the trees that form a population. It is conjectured that the crossover bias directly determines the size distribution of trees in genetic programming. Theorems are presented for the one-generation evolution of visitation length both with and without selection. The connection between path length and visitation length is made explicit.

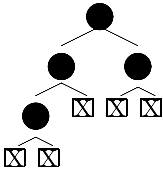
1 Introduction

Nodes in trees are usually distinguished as being either *internal* nodes, or *external* nodes. In genetic programming these are named *functions* and *terminals* respectively. Important characteristics of such trees are the *internal path length*, defined as the sum of path lengths (number of edges) to reach every internal node from the root, and the *external path length*, similarly defined as the sum of path lengths to reach every external node from the root. Many recurrences and generative functions are known for these path lengths for trees of a particular shape, usually binary trees.

In genetic programming, internal nodes and external nodes are simply considered nodes, and operators are defined in terms of such general nodes. The size of a tree is defined as the number of nodes, regardless of them being internal or external. With the notable exception of Koza's 90/10% rule of selecting internal nodes over external nodes, many studies assume that node selection for subtree crossovers and mutations is done by selecting nodes uniformly. This is followed here: the subtree crossover studied here selects nodes uniformly. This is called *standard crossover* here.

In the genetic programming literature, the *shape* of a tree is usually characterized by its *size* and *depth*. Here an alternative measure is examined: a close relative to the total path length of a tree, called the visitation length. The visitation length gives a natural distinction between balanced and skewed trees of

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 33–44, 2007.
© Springer-Verlag Berlin Heidelberg 2007



$\pi(T) = 4$	internal path length
$\zeta(T) = 12$	external path length
$f(T) = 4$	number of internal nodes
$e(T) = 5$	number of external nodes
$s(T) = 9$	size
$\phi(T) = 25$	visitation length

Fig. 1. Example tree and example values for the various functions defined on the tree that are used throughout this paper. Circles are internal nodes and crossed boxes external nodes.

the same size and is directly related to the average size of the subtrees in a tree. A tree's visitation length has a number of important analytical properties that are directly related to the evolution of sizes and shapes in genetic programming. In particular, it will be shown that a crossover bias is present in the evolution of this visitation length. This crossover bias is fully determined by a covariance between sizes and levels in a tree, and it is conjectured that is determines the distribution of sizes in a population of such trees. The paper presents theorems for the evolution of visitation length both with and without selection, alongside some empirical investigations into the effect of crossover bias in genetic programming. This paper presents a mathematical foundation for inquiries into the evolution of tree topology both with and without selection.

2 Sizes and Levels in Trees

Definition 1 (Notation and Size). *Given a tree T, the function $s(T)$ defines the number of nodes (both external and internal) in the tree. To subtree relation is defined through the $\in_s$ symbol: $t \in_s T$ is true if and only if tree t is a subtree of tree T: this defines all subtrees, not only the direct descendants. Uppercase (T) will be used to denote a rooted tree (used for selection and variation), while lowercase (t) is used to designate subtrees. A tree is considered to be a subtree of itself. To sum over all subtrees in a tree, the notation $\sum_{t \in_s T}$ will be used. To determine the direct descendents (immediate subtrees) of a tree T, indexing will be used, using the $c(T)$ to define the number of children of T. Internal and external path length are denoted with π and ζ respectively, while f and e give the number of internal and external nodes.*

Using this notation, the size of a tree can be defined in various ways:

$$s(T) = \sum_{t \in_s T} 1 = 1 + \sum_{i=1}^{c(T)} s(T_i) = f(T) + e(T)$$

Similarly, using the δ function defined to return 1 when the argument is true, and 0 otherwise, the size of a subtree t from T can be defined as the number of subtrees that are a subtree of t:

$$s(t) = \sum_{u \in_s T} \delta(u \in_s t)$$

Definition 2 (Level). *The* level *of a subtree t in a particular tree T is defined as the the number of nodes that need to be traversed to reach the root node, including itself. Thus the level $l(T)$ of a root node equals 1, while the nodes accessible directly from the root will be found at level 2, etc. The level of a subtree t from a tree T can thus be defined as the number of nodes for which t is a subtree:*

$$l(t) = \sum_{u \in_s T} \delta(t \in_s u)$$

Lemma 1 (Symmetry between sizes and levels). *The sum of sizes of all subtrees from T $\sum_{t \in_s T} s(t)$, is equal to the sum of levels $\sum_{t \in_s T} l(t)$.*

Proof. Using Definitions 1 and 2:

$$\sum_{t \in_s T} s(t) = \sum_{t \in_s T} \sum_{u \in_s T} \delta(u \in_s t) = \sum_{t \in_s T} \sum_{u \in_s T} \delta(t \in_s u) = \sum_{t \in_s T} l(t)$$

□

Definition 3 (Total Visitation Length and Mean Subtree Size). *The sum of the sizes (and equivalently by Lemma 1 the sum of levels) has symmetric properties and is directly related to the average subtree size. This sum is denoted here with the function ϕ, and is called the* visitation length*:*

$$\phi(T) = \sum_{t \in_s T} s(T) = \sum_{t \in_s T} l(T) = s(T) + \sum_{i}^{c(T)} \phi(T_i)$$

Using ϕ, both the mean subtree size $\bar{s}$ and mean subtree level $\bar{l}$ of a tree T can be defined as:

$$\bar{s}(T) = \bar{l}(T) = \frac{\sum_{t \in_s T} s(t)}{s(T)} = \frac{\phi(T)}{s(T)}$$

The visitation length ϕ measures the total number of nodes that need to be visited starting at the root. It has a number of important properties, and much of the remainder of this paper is devoted to the study of ϕ. The visitation length, ϕ, is directly related to the *total path length*:

Theorem 1 (Total Path Length). *The* visitation length *ϕ is, for trees, directly related to the total (internal and external) path length through:*

$$\phi(T) = \pi(T) + \zeta(T) + s(T)$$

Proof. The level *of a subtree is the path length to the root plus 1 (Definition 2). For all internal nodes the sum of levels equals $\pi(T) + f(T)$ while for all external nodes to $\zeta(T) + e(T)$. By definition, $s(T) = e(T) + f(T)$.* □

The definitions of *visitation length* and *path length* differ solely in the manner of counting: nodes (vertices) and edges respectively. The functional ϕ as defined on trees has some important properties. For a tree with a given size s, ϕ will take on smaller values, the more balanced the tree is. Figure 2 gives an example of this relation. The visitation length is used by for instance [7] to define an alternative to size based parsimony pressure in order to steer the population to small, balanced, trees.

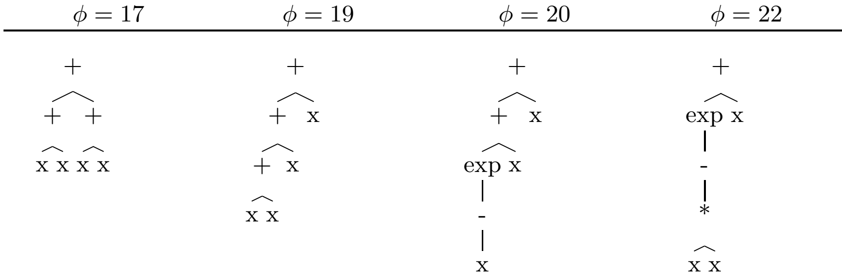


Fig. 2. Total visitation length for a number of differently shaped trees of size 7. The less balanced the tree, the larger the visitation length.

3 Empirical Behaviour of ϕ in Genetic Programming

Although the change in the size of a tree of a population undergoing standard crossover is zero in the expectation, this does not necessarily hold for the visitation length. ϕ is related to the both the shape and the size of the tree. A simple experiment is set up to investigate the behaviour of a population undergoing only crossover, without selection. A population of 5000 binary trees is created through one of three methods: *grow*, *ramped-half-and-half* and *skew*[1], each governed by a single parameter, the maximum depth of the tree. For all methods, when the depth limit of 7 is reached, only terminals are selected.

Figure 3 shows the relationship between the average size of the population and the average visitation length ϕ for a number of runs of genetic programming using different initialization strategies. A clear convergence to a particular relationship between the two variables is observed. The transient can however be long.

4 Analysis

To study the evolution of the visitation length $\bar{\phi}$ without selection, and to shed some light on this apparent convergence to a fixed relationship with the average size, the microscopic mechanics of standard crossover are examined. It is well-known that this operation on a population does not alter the expected size of the population in the next generation. However, this is not necessarily the case for the average visitation length.

[1] *Skew* creates a binary tree with at each node at least one terminal.

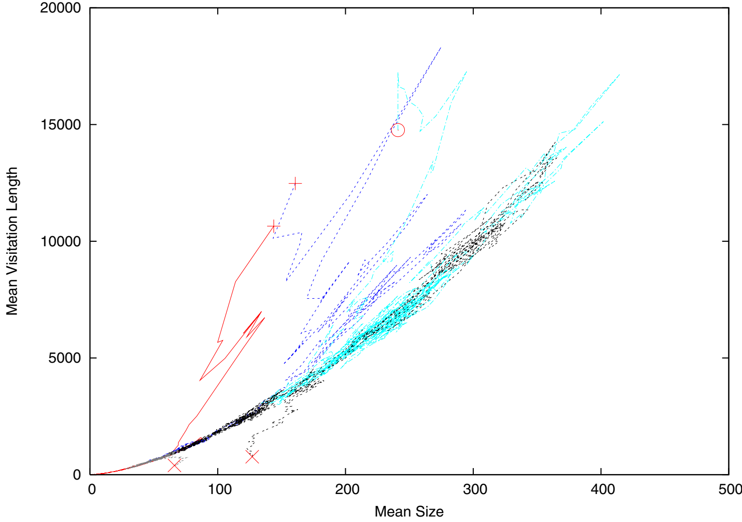


Fig. 3. Phase space plot of $\bar{\phi}$ and $\bar{s}$. Circles, crosses and plusses depict the beginning of runs initialized using the *grow*, *skew*, and *ramped-half-and-half* respectively.

Notation. Below, P will be used to denote a population of n trees. $T \in P$ denotes that the rooted tree T is part of the population, summations such as $\sum_{T \in P}$ denote a summation over all trees (not subtrees) in the population. For summing over all subtrees and levels in a population, the double summation $\sum_{T \in P} \sum_{t \in_s T}$ will be used.

Definition 4 (Size-Level Covariance). *The* size-level *covariance on a population P, is defined as:*

$$Cov_{sl}(P) = \frac{1}{n} \sum_{T \in P} \frac{1}{s(T)} \sum_{t \in_s T} (s(t) - \bar{s}(P))(l(t) - \bar{s}(P))$$

$$= \frac{1}{n} \sum_{T \in P} \frac{\sum_{t \in_s T} s(t) l(t)}{s(T)} - \bar{s}(P)^2$$

With $\bar{s}(P) = \frac{1}{n} \sum_{T \in P} \frac{\phi(T)}{s(T)}$

Note that due to Lemma 1, this average subtree size is equal to the average level. The definition thus defines a true covariance.

Lemma 2 (Microscopic interaction). *The net effect of inserting a single (sub)tree t at the place of u in tree U (denoted by $u \leftarrow t$) in the visitation length $\phi(U)$ is given by:*

$$\phi(U | u \leftarrow t) = \phi(U) - \phi(u) + \phi(t) + (l(u) - 1)(s(t) - s(u))$$

Proof. *Consider the recursion $\phi(U) = s(U) + \sum_i \phi(U_i)$ from Definition 3. For the subtree u from U that is replaced by t, $\Delta\phi = \phi(t) - \phi(u)$. This term is transmitted unaltered in the recursion. The change in size $\Delta s = s(t) - s(u)$ will affect the size of all parents of the node (i.e., all subtrees v from U, for which $u \in_s v$). By definition of the level as one plus the path length to the root node, exactly $l(u) - 1$ ancestors are effected, leading to an additional change in ϕ of $(l(u) - 1)\Delta s$.* □

Theorem 2 (Crossover Bias). *The expected value of the visitation length in a population undergoing standard crossover and without selection is determined by the average visitation length in the current population minus the covariance between sizes and levels of the subtrees in the population.*

$$\bar{\phi}(P') = \bar{\phi}(P) - Cov_{sl}(P)$$

Proof. *Averaging over all pairs of trees in a population P consisting of n trees, and all possible crossover points, using Lemma 2, where the individual terms dependent on ϕ, $1/2 \sum_{T,U \in P}(\phi(T) + \phi(U)) = \bar{\phi}(P)$ and $1/2 \sum_{T,U \in P} \sum_{u \in_s U, T \in_s T}(\phi(t) - \phi(u)) = 0$, leads to a total change in $\phi(P)$ of:*

$$\Delta\phi(P) = \frac{1}{n^2} \sum_{T \in P} \sum_{U \in P} \left[\frac{\sum_{t \in_s T} \sum_{u \in_s U} (l(u) - 1)(s(t) - s(u))}{s(T)s(U)} \right]$$

After some algebra (in Appendix), this reduces to:

$$\Delta\phi(P) = \frac{1}{n^2} \left[\sum_{T \in P} \frac{\sum_{t \in_s T} s(t)}{s(T)} \right]^2 - \frac{1}{n} \sum_{T \in P} \frac{\sum_{t \in_s T} l(t)s(t)}{s(T)} = -Cov_{sl}(P) \qquad \square$$

Although the expected size in the next generation for a population undergoing standard crossover and no selection is expected to be zero, this is not the case for the expected visitation length of the trees composing the population. The size-level covariance $\mathrm{Cov_{sl}}$ introduces a bias in the *expected* visitation length, and evolves this macroscopic quantity toward a region where this bias is no longer present. When $\mathrm{Cov_{sl}} = 0$, this bias disappears. Theorem 2 relates three quantities: the expected value of the visitation length, the average subtree size and the product of sizes and levels.

The main effect that is shown here is the effect on the visitation length of inserting variable sized subtrees at various levels in a tree. This effect is captured in the covariance between sizes and levels in the tree. Lemma 2 shows the microscopic basis of this covariance, and holds for any replacement of a subtree in a tree. Other crossovers and mutations that insert variable length subtrees in various locations in the tree will have a similar derivation associated with them, where the uniform probabilities are replaced by non-uniform probabilities. At this point it is not clear which, if any, operator will have no crossover bias. Analyzing other operators and designing an unbiased operator is left for future work.

The crossover bias is the product of sizes and levels, both compared with the *mean subtree size* in the population. For populations consisting of identical

trees, this quantity appears to be nonzero in general[2]. If this is the case, to make the covariance disappear, the mean subtree size in the population needs to be significantly smaller than the subtree size of the larger trees. To make this happen, small trees need to be sampled more frequently than large trees.

4.1 Binary Trees and the Catalan Distribution

It is conjectured here that the crossover bias is mainly related to the distribution of tree size in the population, *not* to a particular preference for balanced/skewed trees. As is conjectured elsewhere [3], a population not undergoing selection will evolve to the most common shapes. Here it is empirically investigated if shape, as measured by total path length or visitation length, indeed evolves towards their expected (common) values. If so, crossover bias does not affect the shape at all, and its full effect must lie in influencing the distribution of sizes in the population. As short trees have a smaller visitation length than larger trees, the relationship being non-linear, a particular size distribution can have an effect on expected visitation length, without changing expected size.

For binary trees, the expected path lengths under the Catalan distribution can be found in [6], chapter 5:

Theorem 3 (Sedgewick and Flajolet). *For a binary tree T with n* internal *nodes, selected at random with the Catalan distribution*

- $E(\pi(T)) = n\sqrt{\pi n} - 3n + O(\sqrt{n})$
- $E(\zeta(T)) = n\sqrt{\pi n} - n + O(\sqrt{n})$.

And thus,

Corollary 1. *For a binary tree T with $n = (s(T)-1)/2$* internal *nodes, selected at random with the Catalan distribution:*

$$E(\theta(T)) = 2n\sqrt{\pi n} - 2n + 1 + O(\sqrt{n})$$

Proof. Direct by using Theorem 1 in Theorem 3 □

The experiment in Figure 4 indicates that a population undergoing standard crossover apparently samples trees from this Catalan distribution. Given a particular size of a tree, the expected value of ϕ of such trees is determined by the the expected tree shapes, up to a factor of $\sqrt{n}$. The figure shows a close fit given some indication that the conjecture is true, and that no particular shapes are preferred by standard crossover. The main effect of the crossover bias would then be manifested as a preference for a particular limit distribution of tree sizes. Figure 4 also depicts this distribution of sizes for binary trees without selection as a histogram. The experiment exhibits a strong preference for trees of small size. The relationship between crossover bias and size distribution has been examined

[2] No proof is available at this point, it is however experimentally verified for all trees consisting of unary, binary and ternary nodes up to depth 7.

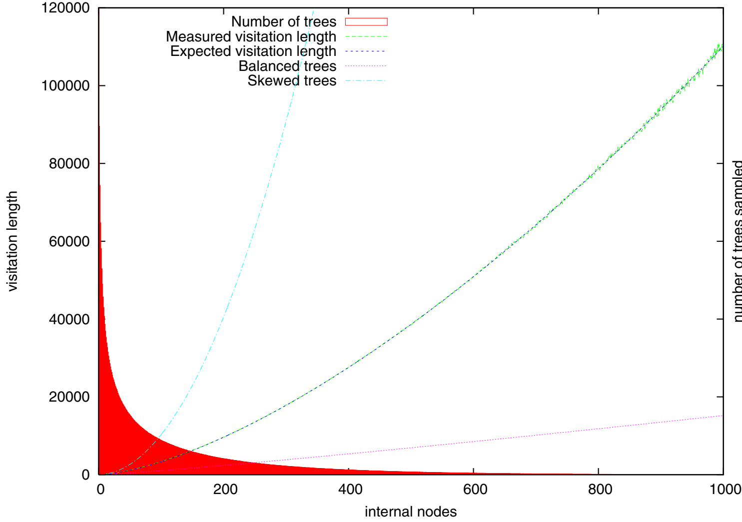


Fig. 4. Visitation length given size, both measured and expected through Corollary 1 for binary trees, *given* a particular size. The graph is the result of 100 runs of 5000 trees, operating without selection. Also shown is the distribution of the sizes of the trees that are sampled, and the visitation length for maximally balanced and skewed trees.

before [4], albeit for unary trees, and led to the discovery of a limit distribution for such unary trees in the form of a gamma distribution. For binary trees, a different distribution is induced, one where small trees, in particular terminals, are over-abundant.

5 Size-Level Covariance and Fitness

Because the effect of crossover on ϕ is non-zero in general, this can be expected to have an effect on the composition of a population undergoing selection. Following Altenberg [1,2] the effect of fitness on an evolving population can be studied using a *canonical* genetic algorithm. The expected value of any measurement function F in such a canonical algorithm can be formulated as:

$$\bar{F}' = \sum_x p(x)' = \sum_x \sum_{y,z \in P} F(x) T(x \leftarrow y, z) \frac{w(y)w(z)}{\bar{w}^2} p(y)p(z) \qquad (1)$$

where $p(x)$ measures the frequency of genotype (tree) x in the next generation, based on current frequencies, fitness w, and the transmission function T that gives the probability of creating genotype x based on y and z. This equation is directly related to Price's Covariance and Selection Theorem [5] (see Altenberg [1] for a complete derivation).

Theorem 4 (Visitation Length and Fitness). *The expected size of the visitation length ϕ in a population undergoing standard crossover and selection over one generation is given by:*

$$\bar{\phi}(P') = \bar{\phi}(P) + \underbrace{Cov(\phi(T), \frac{w(T)}{\bar{w}})}_{Fitness/\phi\ covariance} - \underbrace{\sum \frac{w(T) \sum (s(t) - \bar{s}(P))\,(l(t) - \bar{s}(P))}{\bar{w} s(T)|P|}}_{Crossover\ bias}$$

Proof. *in Appendix* □

This result on visitation length translates directly to total path length:

Corollary 2 (Path length and Fitness). *The evolution of the total path length $p(x) = \pi(x) + \zeta(x)$ under standard crossover and selection is given by:*

$$\bar{p}(P') = \bar{\phi}(P') - \bar{s}(P) - Cov(s(T), w(T)/\bar{w})$$

Proof. This is direct by using Theorems 1 and 4 and Price's Covariance and Selection theorem applied to size. □

As can be expected, the bias induced by standard crossover without selection gets transmitted when using selection. The evolution of visitation length is determined by the crossover bias without selection from Theorem 2, and covariances between both fitness and visitation length, and fitness and the product of size and level of individual trees.

To investigate the effect of crossover bias on a population undergoing selection, an experiment is performed. The problem is defined by a terminal set consisting of the constant 1, a function set consisting of the unary minus operator $-$, and the binary addition operator $+$. The goal of the problem is to find an expression that evaluates to the value 0. The smallest solution for this problem are two trees consisting of 4 nodes: $1 + -1$ and $-1 + 1$. To keep bloat under control and to make the smallest solutions global optima, a small amount of parsimony pressure is used. The population performs proportional selection with crossover only (thus no reproduction), using a population of size $50,000$. The fitness value used in proportional selection is $w(T) = e^{-\text{eval}(T)^2} \times 2^{-1/1000 \times |T|}$. Initialization is *ramped-half-and-half*, and invariably leads to finding the global optimum after initialization. With the global optimum already found, failed runs are effectively eliminated and the transient to the equilibrium state of maximal average fitness can be studied.

Figure 5 depicts the evolution of the population towards equilibrium. Although the optimum is a four-node solution, the mean size of the trees in the population grows. First slowly, but when a critical value of approximately 20 nodes is reached, the population undergoes a rapid growth until at 50 nodes, equilibrium is reached due to parsimony pressure. Elongated runs do not show any growth after this point is reached. Interestingly, bloat does occur, even when there is a fitness disadvantage for larger trees. Depicted is also the balance that is achieved between the covariance of fitness w and visitation length ϕ on the

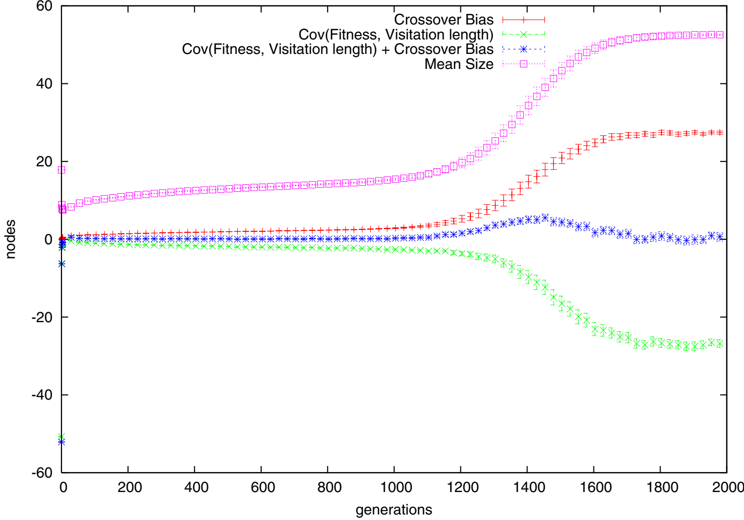


Fig. 5. Transient to the equilibrium for a population that already found a global optimum for 100 runs. Depicted are the mean size of the trees in the population, the covariance between fitness and ϕ alone, the additional terms induced by the crossover bias, and finally the resulting change in ϕ, $\Delta\phi$. 95% confidence intervals are included.

one hand, and the crossover bias on the other. The experiment consistently shows that the fitness function w is correlated with individuals of lower visitation length. In particular, this holds when the population is truly converged and the expected size differential becomes zero.

It can however not be concluded that the fitness function indicates that balanced trees are preferred by the fitness function and that crossover bias hinders this. Although lower visitation length indicates more balanced trees, this only holds for *trees of the same size*. Smaller trees usually have a smaller visitation length than larger trees. This relationship is non-linear: the experimental observation can then simply be an indication that the fitness function prefers a different size distribution, crossover bias hindering this.

The experiment does show however that the crossover bias is not necessarily aligned with the information obtained by measuring fitness. A dynamic balance is found for the expected visitation length $\bar{\phi}$ during the run and the covariance between ϕ and w is counteracted exactly by the size-level covariance. It can be expected that such a conflict of forces has an effect on the optimization capabilities of an evolving population. At this point, a clear view on the exact effect of the crossover bias in an evolving population is unavailable.

6 Conclusion

This work examines the one generation evolution for path length and visitation length in genetic programming for standard crossover, both with and without

selection. A *crossover bias* is derived that works against the covariance between fitness and visitation length. The crossover bias takes the form of a covariance between the sizes and levels in the tree. Theorems are presented for the exact effect this crossover bias has on the evolution of shape, both with and without selection.

It is hypothesized that this crossover bias mainly introduces a preference for a particular size distribution as empirical evidence indicates that no preference for a particular visitation length is induced by crossover. The paper gives a theoretical foundation for further inquiry into these matters.

References

1. L. Altenberg. The evolution of evolvability in genetic programming. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 3, pages 47–74. MIT Press, 1994.
2. L. Altenberg. The Schema Theorem and Price's Theorem. In L. D. Whitley and M. D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 23–49, Estes Park, Colorado, USA, 31 July–2 Aug. 1994 1995. Morgan Kaufmann.
3. W. B. Langdon, T. Soule, R. Poli, and J. A. Foster. The evolution of size and shape. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA, June 1999.
4. N. F. McPhee and R. Poli. A schema theory analysis of the evolution of size in genetic programming with linear representations. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 108–125, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.
5. G. Price. Selection and covariance. *Nature*, 227:520–521, 1970.
6. R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. AddisonWesley, 1996.
7. G. Smits and M. Kotanchek. Pareto-front exploitation in symbolic regression. In U.-M. O'Reilly, T. Yu, R. L. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 17. Kluwer, Ann Arbor, 13-15 May 2004.

Appendix

Proof of Theorem 2

$$\begin{aligned}
\Delta\phi(P) &= \frac{1}{n^2}\sum_{T\in P}\sum_{U\in P}\left[\frac{1}{s(T)s(U)}\sum_{t\in_s T}\sum_{u\in_s U}(l(u)-1)(s(t)-s(u))\right]\\
&= \frac{1}{n^2}\sum_{T\in P}\sum_{U\in P}\frac{1}{s(T)s(U)}\sum_{t\in_s T}\sum_{u\in_s U}[s(t)l(u)-l(u)s(u)]\\
&= \frac{1}{n^2}\sum_{T\in P}\sum_{U\in P}\frac{1}{s(T)s(U)}\left[\sum_{t\in_s T}\sum_{u\in_s U}[s(t)l(u)]-s(T)\sum_{u\in_s U}l(u)s(u)\right]\\
&= \frac{1}{n^2}\sum_{T\in P}\sum_{U\in P}\frac{\sum_{t\in_s T}s(t)\sum_{u\in_s U}l(u)}{s(T)s(U)}-\frac{\sum_{u\in_s U}l(u)s(u)}{s(U)}\\
&= \frac{1}{n^2}\sum_{T\in P}\sum_{U\in P}\frac{\sum_{t\in_s T}s(t)\sum_{u\in_s U}s(u)}{s(T)s(U)}-\frac{1}{n}\sum_{U\in P}\frac{\sum_{u\in_s U}l(u)s(u)}{s(U)}\\
&= \frac{1}{n^2}\left[\sum_{T\in P}\frac{\sum_{t\in_s T}s(t)}{s(T)}\right]^2-\frac{1}{n}\sum_{T\in P}\frac{\sum_{t\in_s T}l(t)s(t)}{s(T)}
\end{aligned}$$

Proof of Theorem 4 Insert Theorem 2, into Equation 1:

$$\begin{aligned}
\bar{\phi}(P') &= 1/2\sum_{T,U\in P}w(T)w(U)/\bar{w}^2\left[\phi(T\cup U)+\mathrm{Cov}_{\mathrm{sl}}(T\cup U)\right]\\
&= 1/|P|\sum_{T\in P}w(T)/\bar{w}\phi(T)+\\
&\quad 1/2\sum_{T,U\in P}\frac{w(T)}{s(T)\bar{w}}\sum_{t\in_s T}(s(t)-\bar{s}(T\cup U))(l(t)-\bar{s}(T\cup U))+\\
&\quad \frac{w(U)}{s(U)\bar{w}}\sum_{u\in_s U}(s(u)-\bar{s}(T\cup U))(l(u)-\bar{s}(T\cup U))\\
&= \bar{\phi}(P)+\mathrm{Cov}(\phi(T),w(T)/\bar{w})+\sum\frac{w(T)\sum\left(s(t)-\bar{s}(P)\right)\left(l(t)-\bar{s}(P)\right)}{\bar{w}s(T)|P|}
\end{aligned}$$

Density Estimation with Genetic Programming for Inverse Problem Solving

Michael Defoin Platel[1,2], Sébastien Vérel[2], Manuel Clergue[2], and Malik Chami[1]

[1] Laboratoire I3S
CNRS-Université de Nice Sophia Antipolis, France
[2] Laboratoire d'Océanographie de Villefranche sur Mer
CNRS-Université Pierre et Marie Curie-Paris6, France

Abstract. This paper addresses the resolution, by Genetic Programming (GP) methods, of ambiguous inverse problems, where for a single input, many outputs can be expected. We propose two approaches to tackle this kind of many-to-one inversion problems, each of them based on the estimation, by a team of predictors, of a probability density of the expected outputs. In the first one, Stochastic Realisation GP, the predictors outputs are considered as the realisations of an unknown random variable which distribution should approach the expected one. The second one, Mixture Density GP, directly models the expected distribution by the mean of a Gaussian mixture model, for which genetic programming has to find the parameters. Encouraging results are obtained on four test problems of different difficulty, exhibiting the interests of such methods.

1 Introduction

One of the main application of Genetic Programming (GP) is for the approximation of unknown functions, a task known as Symbolic Regression [7]. To construct a such approximator, a GP system is trained on a given dataset that consists of pairs of inputs and desired outputs, representative of an unknown function. This is a direct problem.

While a direct problem describes a Cause-Effect relationship, an Inverse Problem (IP) consists in retrieving the causes responsible of some observed effects. For example, inferring gene regulatory networks from gene activity data or deriving some water constituants from the ocean colour are actually IP. GP has already been introduced as a method for solving IP, such as in [4,3,11,5]. However, IP are often far more difficult to solve than direct problems, since different causes may produce the same effect, *i.e.* the solution of an IP may be not unique. In that case, the IP is said to be redundant or ambiguous or, in a more formal way, ill-posed[1]. In the context of learning from datasets, a redundant IP corresponds to a Many-To-One mapping inversion, *i.e.* to a given input y_i, a set of outputs $\mathcal{X}_i$ is expected. The purpose of this study is to enhance the inversion

[1] In fact, there are three sufficient conditions for ill-posedness which are the existence, the continuity and the redundancy of the solutions.

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 45–54, 2007.
© Springer-Verlag Berlin Heidelberg 2007

of Many-To-One mappings with GP. Here, the set of expected outputs $\mathcal{X}_i$ is seen as a set of realisations of a random variable with an unknown probability density which has to be estimated. We propose two different ways to estimate the probability density of these answers and both are based on the possibility of producing multiple outputs with GP.

The section 2 highlights the limits of the classical Symbolic Regression approach for ambiguous IP and reviews the possibilities of producing multiple outputs with GP. Then, two original methods to tackle the redundancy problem are proposed in section 3 and tested on four benchmark problems in section 4. Finally, the application of this work and the possible further developments are discussed in the conclusion.

2 Inverse Problem and GP

2.1 Learning with Redundancy

With few hypothesis on the instructions set used to build programs, GP is an universal approximator [13] that can learn an arbitrary target function $t : X \mapsto Y$ from a dataset $D_t = \{(x_1, y_1) \dots (x_N, y_N)\}$. Here, we consider the pairs (x_i, y_i) as the realisations of two random variables $\mathbf{x}$ and $\mathbf{y}$ and we note f the function implemented by a GP program. When training a GP system to approximate an unknown mapping defined by D_t, an error function, such as for example the classical Root-Mean-Square Error, is minimized. It is known that in the theoretical case, this process leads to find the optimal answer $f^*(\mathbf{x})$ which is the conditional mean $E(\mathbf{y}|\mathbf{x})$, see [1] for details.

When solving an IP, the dataset $\overline{D_t}$ used can always be seen as the set of the N reversed pairs (y_i, x_i). If the direct function t is not injective, the learning of $\overline{D_t}$ corresponds to a Many-To-One mapping inversion. Hence, for each y_i a set of of outputs $\mathcal{X}_i$ is expected, and the single answer $f(y_i)$ given by GP can be very poorly adapted. Indeed, during the training phase, f tends to converge towards the theoretical optimal answer $f^*(\mathbf{x}) = E(\mathbf{x}|\mathbf{y})$, which is, in the better case, only one of the expected answer.

To illustrate this, we have created a dataset D_d from the target function $d : \Re \mapsto \Re$ such that $y = sin(x^2) + \epsilon$, with ϵ a random variable with normal distribution $\mathcal{N}(0, 0.2)$. An (nearly good) approximation of the corresponding theoretical $f^*(\mathbf{x})$ is given by a standard GP system from D_d with $N = 500$ learning examples in the range $[-2, 2]$, see Figure 1. In a same way, in Figure 2, the output of GP corresponding to the inversion of d is plotted[2]. We can see that GP has produced a very unstable function that highly overfits the dataset $\overline{D_d}$.

2.2 Multiple Outputs

In the GP field, several studies are related to IP solving (see for example [4,3,11,5]) but very few of them have investigated the question of the redundancy. However a

[2] The evolutionary parameters settings of the direct case were kept here.

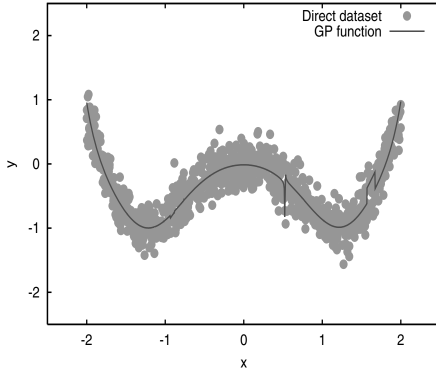


Fig. 1. Example of direct problem solving with Symbolic Regression

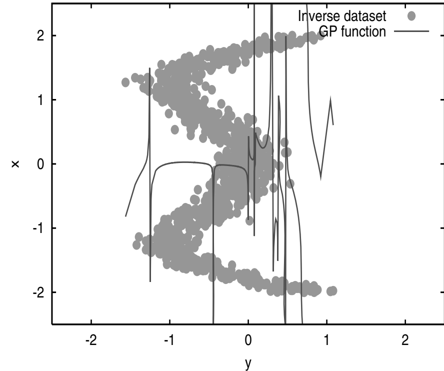


Fig. 2. Example of inverse problem solving with Symbolic Regression

noticeable exception can be found in [9] and will be discussed further. To overcome the non-uniqueness of the solution, instead of predicting the conditional mean, one way is to ensure that the output of an inverse model is at least one of the expected solutions [6]. A second idea is to produce a reduced list of illustrative examples [8] and a third possibility consists in approximating the distribution of the plausible solutions, that is the conditional probability density $p(\mathbf{x}|\mathbf{y})$ as explained in [1]. In this study, we are interested in the latter two possibilities, namely, those that require for GP to produce multiple outputs.

A lot of previous work in GP deals with multiple outputs. Probably the most straightforward way of doing this is by implementing sophisticated systems evolving programs able to manipulate a complex data structure such as a vector or a matrix [10]. Thus, one single GP program is responsible of producing multiple outputs. In [9], a technique based on boosting, that usually creates *a posteriori* mixtures of potential solutions, was extended to handle the ambiguity problem. Similarly, with the *Parisian* approach, used also to tackle an IP in [4], a subset of the population is selected to build the final answer. In these two previous examples, some GP programs are optionally associated to form a set of multiple outputs. Conversely, it is also possible to definitely link together several programs as being the co-operative members of a team and then, to make the whole team evolve [2,5]. In what follows, we will see how the multiple outputs of a team of programs can be used to estimate a probability density.

3 Density Estimation with GP

3.1 From Team to Probability Distribution

Practically, to solve an IP, a dataset $\overline{D}$ consisting of pairs (y_i, x_i) is used and for a given input y_i, the goal is to approximate the distribution of the associated outputs $\mathcal{X}_i$. In the statistical inversion theory (see [12]), the pairs (y_i, x_i) are

considered as the joint realisations of two random variables $\mathbf{y}$ and $\mathbf{x}$ and the solution of an IP has the form of a conditional probability density $p(\mathbf{x}|\mathbf{y})$. We propose two different ways to approximate the subsequent distribution.

In the first approach, called Stochastic Realisation GP (SR-GP), for each input y_i, we consider the n outputs of a GP team T as the n realisations of an unknown random variable. We note f_j, the function implemented by the jth member of T. With SR-GP, the evolutionary system try to find teams which outputs $f_j(y_i)$ report distributions similar to the distributions $p(\mathbf{x}|\mathbf{y} = y_i)$ for all the y_i of the dataset.

The second approach is called Mixture Density GP (MD-GP). Here, a parametric model, namely the finite Gaussian Mixture Model (GMM), is used as explained in [1]. An unknown density $p(\mathbf{x}|\mathbf{y})$ can always be represented as a finite sum of G Gaussian densities such as :

$$p(\mathbf{x}|\mathbf{y}) = \sum_{g=1}^{G} w_g(\mathbf{y})\ \phi_g(\mathbf{x}|\mathbf{y})$$

with $\phi_g(\mathbf{x}|\mathbf{y})$ a normal density $\mathcal{N}(\mu_g(\mathbf{y}), \sigma_g(\mathbf{y}))$. In MD-GP, each of the $n = 3 \times G$ members of a team T approximates one of the functions ϕ_g, μ_g and σ_g that actually tune the GMM. It is worth noticing that, except for the means μ_g, the parameters of a GMM are constrained for a given y_i, since the deviations $\sigma_g(y_i)$ are positive real numbers, and since the weights $w_g(y_i)$ are also positive numbers but with $\sum w_g(y_i) = 1$. We note W and S, two functions that transform the team outputs into respectively valid $w_g(y_i)$ and $\sigma_g(y_i)$ parameters for GMM[3]. So, for a given input y_i, the answer of a team T is :

$$T(y_i) = \left\{ \begin{array}{lllll} w_1(y_i) = W(\ f_1(y_i)\ ) & \ldots & w_G(y_i) = W(\ f_{n-2}(y_i)\ ) \\ \mu_1(y_i) = \quad f_2(y_i) & \ldots & \mu_G(y_i) = \quad f_{n-1}(y_i) \\ \sigma_1(y_i) = S(\ f_3(y_i)\ ) & \ldots & \sigma_G(y_i) = S(\ f_n\ (y_i)\ ) \end{array} \right\}$$

Hence $T(y_i)$ is directly used to tune the GMM from which a set of r realisations can be produced (with usually $r >> n$). With MD-GP, the evolutionary system tries to find teams tuning GMM, so that the GMM realisations according to y_i report distributions similar to the distributions $p(\mathbf{x}|\mathbf{y} = y_i)$.

Intuitively, we understand that in SR-GP, a huge number of parameters have to be retrieved, since the size of the teams have to be big enough to produce a sufficient number of realisations (probably more than 10^3) so that the subsequent distributions can be significantly tested. However, one can presume that with this representation, the search space is "smooth" since the modification of one team member only affects one realisation and so slightly modify the distribution. At the contrary, with MD-GP, fewer parameters have to be retrieved to properly tune the GMM (less than 30 in this paper) and so to produce significant results but it is clear that the modification of only one team member can induce important consequences on the fitness of the whole team.

[3] In this paper, $S(\sigma_g(y_i))$ is simply the absolute value $|\sigma_g(y_i)|$ and similarly $W(w_g(y_i)) = |w_g(y_i)|/\sum |w_g(y_i)|$.

3.2 Construction of Target Distributions

The conditional probability density $p(\mathbf{x}|\mathbf{y})$ is unknown and only pairs (y_i, x_i) are available in a dataset $\overline{D}$. However, illustrative training distributions are required to properly educate the GP system. So for each input y_i, we have to build the probability distribution $p(\mathbf{x}|\mathbf{y} = y_i)$. Our idea is that even if a given value y_i is more likely present at most once in the dataset, many comparable values can be found. Thus here, for each value y_i, a set of k-nearest neighbours $\{y_i \dots y_j\}$ is computed. Then, for each value y_i of the dataset, the set $\{x_i \dots x_j\}$ is turn into binned data, by grouping the events into C specific ranges and so binned distributions are computed. The underlying assumption being that the distribution of the $\{x_i \dots x_j\}$ is similar to $p(\mathbf{x}|\mathbf{y} = y_i)$.

The dataset $\overline{D_d}$ presented section 2.1 is extended to hold $N = 10^5$ pairs and in Figure 3, we have plotted three sets of the $k = 1000$ pairs $(y_i, x_i) \dots (y_j, x_j)$ corresponding to $y_i = -0.93$, -0.19 and 0.49. In Figure 3, three binned distributions are drawn for $C = 50$. Each of them represents an approximation of the theoretical conditional probability $p(\mathbf{x}|\mathbf{y} = y_i)$ for y_i=-0.93, -0.19 and 0.49.

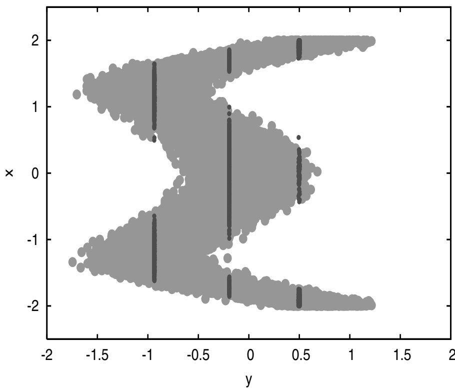


Fig. 3. Examples of pairs (y_i, x_i) corresponding to the k-nearest neighbours of y_i=-0.93, -0.19 and 0.49

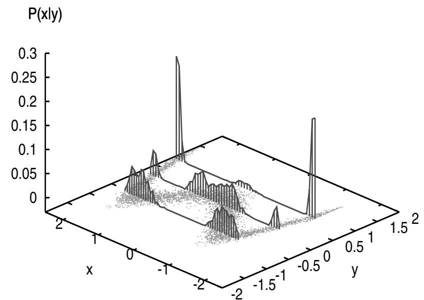


Fig. 4. Examples of binned distributions used to approximate $p(\mathbf{x}|\mathbf{y} = y_i)$, with y_i=-0.93, -0.19 and 0.49

3.3 Fitness Function

In this paper, only the fitness of the teams are computed and no effort have been made to estimate the fitness of the team members. Either with `SR-GP` or `MD-GP`, a team will be considered as a good team, if the set of the realisations produced for each y_i tested report a binned distribution comparable to the binned distribution obtained from a dataset $\overline{D}$, as explained above. To measure the distance between distributions, ϕ-divergences can be used, as for example the chi-square distance. Let be $U(y_i)$ the expected binned distribution computed from $\overline{D}$ for a given y_i and $V(y_i)$, the distribution produced by a team of programs for the same y_i.

We note $U_c(y_i)$ and $V_c(y_i)$, the number of events in the bin c of the two distributions. The total number of events $\#U(y_i)$ is actually independent of y_i in this study and corresponds to the value k defined in section 3.2. Here, the partial fitness of a team T for one sample y_i is

$$E_T(y_i) = \frac{1}{\#U(y_i)} \sum_{c=1}^{C} \frac{(U_c(y_i) - V_c(y_i))^2}{U_c(y_i) + V_c(y_i)}$$

and the total fitness is simply the average of the $E_T(y_i)$ for all the y_i of a dataset $\overline{D}$.

4 Experiments

In this section, we aim to verify the ability of the `SR-GP` and `MD-GP` systems to approximate conditional probabilities and so to solve an IP. The linear stack-based GP implementation described in [5] is used to run the experiments but any other GP implementation suitable for Symbolic Regression can be used instead. The number of members in the teams is static and fixed *a priori*, see [2] for details. Different settings for the fitness function and the genetic operators but also various options for the representation and the conversion of teams are investigated. For each experiment, we perform 30 independent runs and a statistical unpaired, two-tailed t-test with 95% confidence determines if results are significantly different. Populations of teams are randomly created and the maximum creation size of the teams members is 50. The instruction set contains: the four arithmetic instructions `ADD`, `SUB`, `MUL`, `DIV`, the input variable `Y` and one stack-based GP specific instruction DUP which duplicates the top of the operand stack. We add also into the instructions set, two Ephemeral Random Constants noted ERC1 and ERC2, as described in [7], respectively in the ranges $[-100, 100]$ and $[-1, 1]$. The evolution is achieved with elitism, 4-tournament selection and steady-state replacement. Recombination is performed by the standard crossover operator with a rate of 0.7 and mutation with a rate of 1.0, meaning that each program involved in reproduction will undergo, on average, either one insertion or one deletion or one substitution.

Four problems P_a, P_b, P_c, P_d are tested. The corresponding datasets consist of $N = 10^5$ samples from which 100 binned distributions are computed as explained in section 3.2 and 80 are dedicated to the training phase. In fact, the two first test problems P_a and P_b do not correspond to the inversion of a direct function, but for P_a, the GP system has to approximate a bimodal distribution[4] $\frac{1}{2}\mathcal{N}\left(\frac{1}{10}{y_i}^3 - {y_i}^2 + y_i + 9,\ 0.1\right) + \frac{1}{2}\mathcal{N}\left(5({y_i}^3 - 3{y_i}^2) * e^{-y_i} - 3,\ 0.1\right)$ and for P_b, for each input y_i we have $\mathcal{N}\left(y_i, 2.3{y_i}^2 - 1.7y_i - 5.4\right)$. With an increasing difficulty, P_c and P_d are true IP that correspond respectively to the inversion of the function $c(x) = x + 0.3sin(2\pi x) + \epsilon$ with ϵ, a random variable with uniform distribution in the range $[-0.1, 0.1]$ and to the inversion the function d described

[4] This problem is strongly inspired by the work of Paris et al. [9]

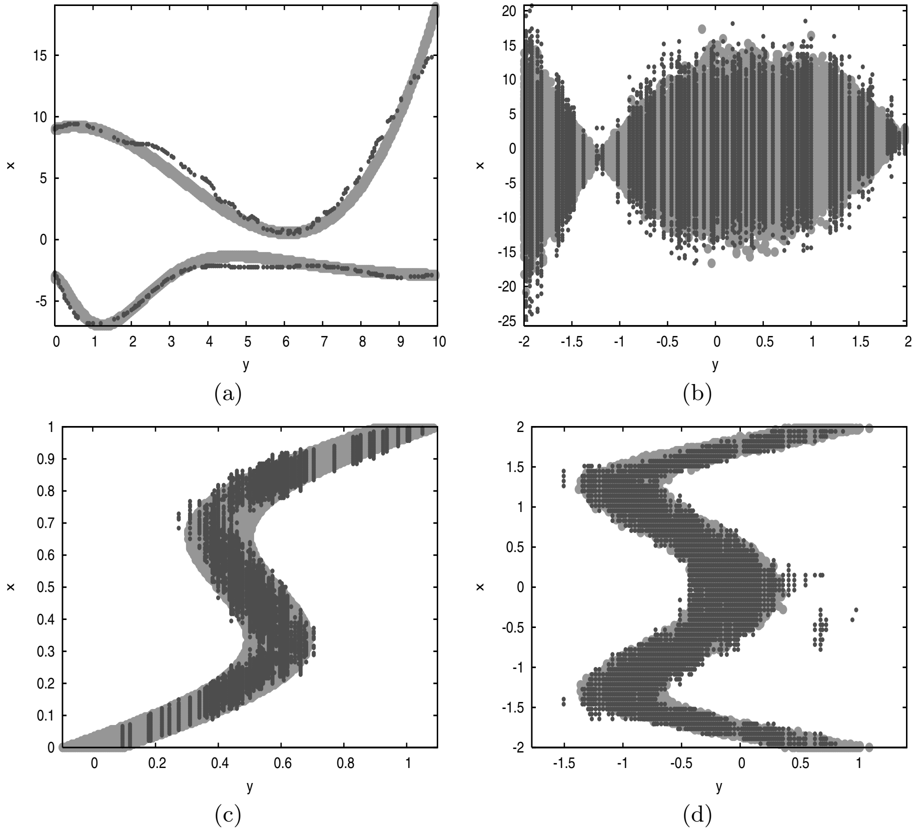


Fig. 5. Density estimation : four test problems

Table 1. Average fitness of the best programs found on the four test problems

GP Type	Pop. Size	Team Size	Max Size	No Gen.	k Real.	Train Fit.	Test Fit.
Problem P_a							
MD-GP	9.10^2	2×3	2.10^2	10^2	10	$0.43_{\sigma=0.16}$	$0.49_{\sigma=0.24}$
SR-GP		2				$\mathbf{0.22}_{\sigma=0.09}$	$\mathbf{0.25}_{\sigma=0.09}$
Problem P_b							
MD-GP	9.10^2	4×3	10^2	10^2	10^3	$\mathbf{0.21}_{\sigma=0.11}$	$\mathbf{0.23}_{\sigma=0.14}$
SR-GP	10^4	10^3				$1.24_{\sigma=0.21}$	$1.43_{\sigma=0.34}$
Problem P_c							
MD-GP	9.10^2	5×3	10^2	2.10^2	10^3	$\mathbf{0.55}_{\sigma=0.11}$	$\mathbf{0.58}_{\sigma=0.14}$
SR-GP	10^4	10^3				$1.02_{\sigma=0.18}$	$1.03_{\sigma=0.24}$
Problem P_d							
MD-GP	9.10^2	6×3	10^2	10^2	10^3	$\mathbf{0.28}_{\sigma=0.06}$	$\mathbf{0.35}_{\sigma=0.07}$
SR-GP	10^4	10^3		10^4		$0.42_{\sigma=0.15}$	$0.51_{\sigma=0.14}$

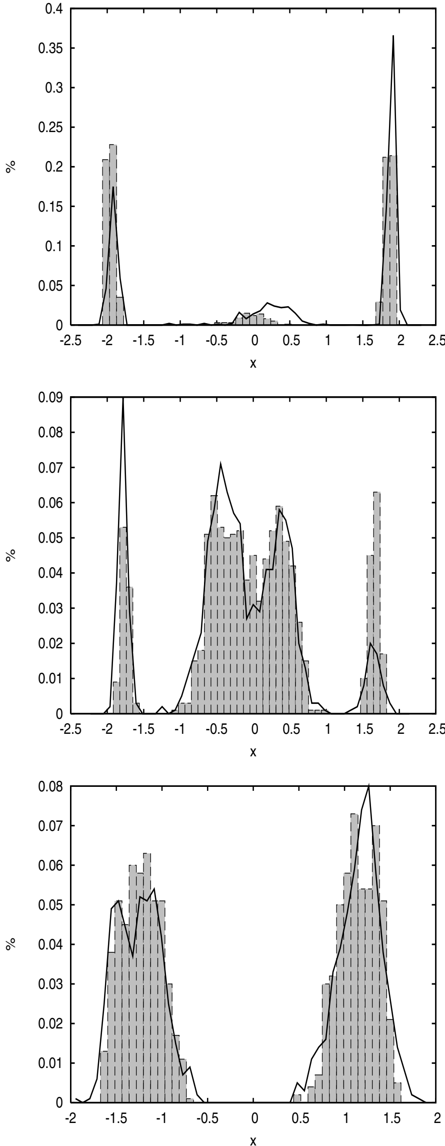


Fig. 6. Examples of density $p(\mathbf{x}|\mathbf{y} = y_i)$ estimation with `MD-GP` for the problem P_d with, from top to bottom, $y_i = 0.49$, -0.93 and -0.19

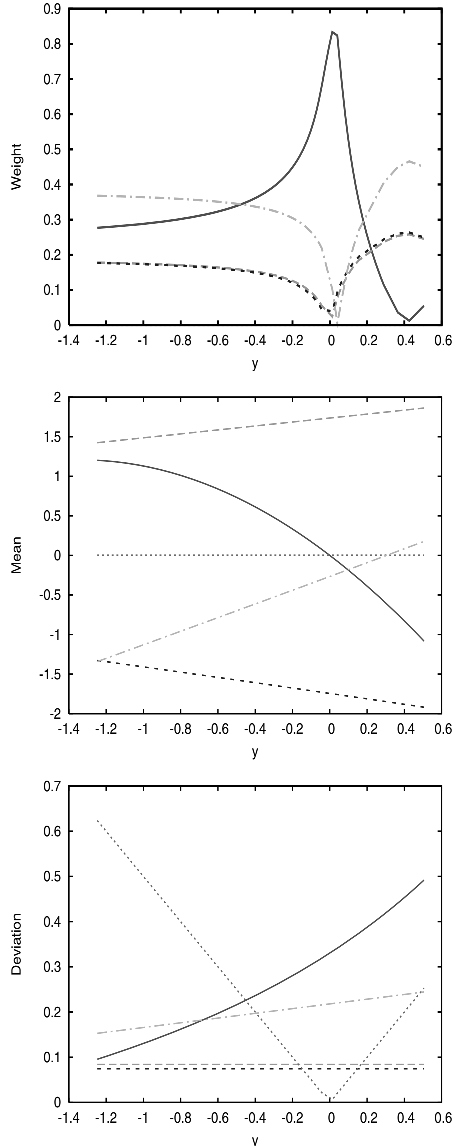


Fig. 7. Example of Gaussian Mixture Model settings as a function of y with $G = 5$ for the problem P_d

section 2.1. The training samples of the four test problems, are plotted (gray points) in Figure 5, respectively part (a), (b), (c) and (d). The outputs (realisations) of representative of the best density estimations performed by our GP systems are also superimposed (in black).

We note from Table 1 that SR-GP is the best approach for tackling P_a even if we allow more generations to SR-GP. Actually, solving P_a consists more in retrieving two different functions than in a density estimation, and we think this is exactly the reason why SR-GP is more adapted. Conversely, MD-GP performs better on the three others problems. To figure out the quality of density estimation given by MD-GP, we have plotted for P_d, three examples of expected and retrieved distributions corresponding to $y_i = 0.49$, -0.93 and -0.19, (see fig. 6), and we think that a good agreement has been obtained. The retrieved distributions are produced by three Gaussian mixtures with $G = 5$. The outputs of the corresponding GP team, consisting of 15 programs, is plotted in Figure 7. We can see (middle part), the expected 'Σ' shape realized by the 5 functions μ_g controlling the means of the Gaussian. Actually, the unwanted null-function has no influence in the model outputs since the corresponding weight Φ_g in the mixture is also null in the top part of Figure 7. We note that there are two different ways of almost "switching-off" a Gaussian since fixing a deviation σ_g to zero is another possible option for the system. This possibility should also be investigated for the approximation of non-continuous function with teams.

5 Conclusion and Perspectives

In many scientific domains, solving IP is now a major concern and the question of their redundancy requires particular answers. A wide range of methods can be used. Often based on a Bayesian approach, they actually report very good performances compared the two GP variants presented here, so that no inter-comparisons were made. However, this paper is a promising proof of concept. Indeed, we note that Symbolic Regression has been introduced probably more as a benchmark problem than as a true potential application for GP, but after more than a decade of improvements, the best programs found by GP are now competitive with others methods, which is very encouraging for SR-GP and MD-GP.

One of the difficulty in comparing with published work is that, for the two alternatives presented here, the fitness function used is a ϕ-divergence, very useful for computing distance between two distributions but unusual in GP. As far as we know, the only previous GP work addressing explicitly the redundancy problem [9], is much more appropriated for functions decomposition, as the problem P_a, than for more complex IP. Our SR-GP system can also easily solve P_a, while MD-GP is much more suitable to tackle IP where complex densities have to be estimated. So a broad variety of redundant IP can be addressed with SR-GP and MD-GP and we think that, for a given IP, preliminary statistical analysis of the dataset, will help to decide which method is adapted but also to *a priori* tune the systems parameters. Moreover, a lot of improvements can be made. We think that further work should address the possibility of : assigning a partial fitness to the team members, producing a variable number of realisations according to the inputs and designing genetic operators appropriate to teams, in particular to teams which members have different semantics in the system as the weights, the means and the deviations of the Gaussian Mixture Model.

References

1. C. M. Bishop. Mixture density networks. Technical report, Aston University, 1994.
2. M. Brameier and W. Banzhaf. Evolving teams of predictors with linear genetic programming. *Genetic Programming and Evolvable Machines*, 2(4):381–407, 2001.
3. M. Chami and D. Robilliard. Inversion of oceanic constituents in case i and case ii waters with genetic programming algorithms. *Applied Optics*, 40(30):6260–6275, 2002.
4. P. Collet, E. Lutton, F. Raynal, and M. Schoenauer. Polar IFS+parisian genetic programming=efficient IFS inverse problem solving. *Genetic Programming and Evolvable Machines*, 1(4):339–361, 2000.
5. M. Defoin Platel, M. Chami, M. Clergue, and P. Collard. Teams of genetic predictors for inverse problem solving. In *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 341–350, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.
6. M. I. Jordan and D. E. Rumelhart. Forward models: Supervised learning with a distal teacher. *Cognitive Science*, 16:307–354, 1992.
7. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.
8. R. Morishita, H. Imade, I. Ono, N. Ono, and M. Okamoto. Finding multiple solutions based on an evolutionary algorithm for inference of genetic networks by s-system. In *Congress on Evolutionary Computation, 2003. CEC '03*, pages 615–622, 2003.
9. G. Paris, D. Robilliard, and C. Fonlupt. Genetic programming with boosting for ambiguities in regression problems. In *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 183–193, Essex, 14-16 April 2003. Springer-Verlag.
10. L. Spector. Autoconstructive evolution: Push, pushGP, and pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.
11. F. Streichert, H. Planatscher, C. Spieth, H. Ulmer, and A. Zell. Comparing genetic programming and evolution strategies on inferring gene regulatory networks. In *Genetic and Evolutionary Computation – GECCO-2004, Part I*, pages 471–480, Seattle, WA, USA, 2004.
12. A. Tarantola. *Inverse Problem Theory: Methods for Data Fitting and Model Parameter Estimation.* Elsevier, Amsterdam, Netherlands, 1987.
13. Xin Yao. Universal approximation by genetic programming. In *Foundations of Genetic Programming*, Orlando, Florida, USA, 13 1999.

Empirical Analysis of GP Tree-Fragments

Will Smart, Peter Andreae, and Mengjie Zhang

School of Mathematics, Statistics and Computer Science,
Victoria University of Wellington, PO Box 600, Wellington, New Zealand
smartwill@mcs.vuw.ac.nz

Abstract. Researchers have attempted to explain the power of Genetic Programming (GP) search using various notions of schema. However empirical studies of schemas have been limited due to their vast numbers in typical populations. This paper addresses the problem of analyzing schemas represented by tree-fragments. It describes a new efficient way of representing the huge sets of fragments in a population of GP programs and presents an algorithm to find all fragments using this efficient representation. Using this algorithm, the paper presents an initial analysis of fragments in populations of up to 300 programs, each up to seven nodes deep. The analysis demonstrates a surprisingly large variation in the numbers of fragments through evolution and a non-monotonic rise in the most useful fragments. With his method, empirical investigation of the GP building block hypothesis and schema theory in realistic sized GP systems becomes possible.

1 Introduction

Since the original work done by Holland [1], many researchers have sought to explain the power of evolutionary search using the notion of schemas and their propagation through evolution. A schema is a set of points in search space that share some syntactic characteristics; in a population of genetic programs, a schema refers to a set of programs that have some common genetic material.

A schema may be an useful way to characterize sets of programs if we assume that programs that share significant amounts of genetic material are likely to have similar outputs and fitnesses. *Good schemas* are those where the common material predisposes programs in the schema to have good fitness; finding these good schemas may allow us to narrow the GP search space to those programs likely to have better fitness.

Further, study of schemas may give us insight into genetic search. In 1975, Holland [1] proposed that, while searching for a good chromosome to solve a problem, the genetic search of Genetic Algorithms (GAs) was also performing a larger search for good schemas, a process he called *inherent parallelism*. Later, Goldberg [2] proposed his *building block hypothesis* (BBH) predicting the propagation of small, fit schemas he called building blocks. While this foundation research dealt with GAs, there has been much research attempting to transfer the results to GP. However, very little of this research empirically analyzes

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 55–67, 2007.
© Springer-Verlag Berlin Heidelberg 2007

practical GP evolutions. The reason for this is simple: when using useful notions of GP schema, a typical population of programs contains a massive number of individual schemas, and naive methods to identify them all are often doomed to failure. Thus, empirical analysis of GP schemas has almost exclusively been restricted to small populations and/or programs.

This paper describes a new tool which vastly expands the ability of GP researchers to empirically analyze fragment schemas in real-sized populations. The tool divides the space of all fragment schemas into groups in such a way that the size of the problem is reduced without unnecessarily restricting analysis.

The paper proceeds as follows: the following section presents an overview of related research, section 3 presents definitions used in the paper, section 4 describes the concept of *maximal fragment*, section 5 introduces a new algorithm which is used in section 6 to analyze GP evolutions, finally section 7 presents conclusions and directions for future research.

2 Related Work

2.1 Schema Theory and Analysis in GAs and GP

In the 1970s, researchers looked to explain the power of GAs. A persuasive argument was presented by John Holland in [1]: while searching for good chromosomes, GAs also perform a parallel meta-search of the patterns common to chromosomes; these patterns he called *schemata* (*schemas* in this document).

When a chromosome is assessed for fitness, doing so obtains new information about the fitnesses of all schemas the chromosome is in. This simultaneous processing of a large number of schemas was given the name *inherent parallelism*. Holland [1] presented the *GAs Schema Theorem*, which gives a lower bound on the expected number of chromosomes belonging to a schema in the next generation in terms of information available in the current generation.

Research applying the GAs schema theorem to GP started in the early 1990's (for example [3,4,5,6,7]). There are few reports of empirical studies of schemas; even in a single small population of small programs the number of distinct schemas is potentially massive, making tracking and analyzing all schemas in all but trivial populations difficult. Some work in the area includes Rosca's extraction of the values of important terms in his schema theorem from actual populations [6]. In 1997, Poli and Langdon [8] performed empirical experiments, tracking creation and transmission of all hyperschemas in populations of fifty boolean programs limited to three or four nodes deep. Langdon and Banzhaf [9] presented an analysis of schema repetition in best-of-run genetic programs from evolutions on two benchmark problems. Wilson and Heywood [10] built on the work of Langdon and Banzhaf, analyzing repeated blocks of instructions in linear genetic programs. Majeed [11] constructed fragments by generalizing those subtrees, of a set maximum depth, that occurred in at least half of the population in the last generation of evolution.

2.2 The Building Block Hypothesis

Building on Holland's research, Goldberg [2] proposed his ambitious *Building Block Hypothesis* (BBH) which predicts the propagation of so-called *building blocks*, and proposes that GAs work in part by combining these building blocks into larger schemas of potentially higher fitness. The theorem proved highly controversial.

Researchers (for example [4,12,13,14]) have analyzed the predictions of the BBH when applied to the new representation of individuals in GP. There are reasons to believe building blocks propagate in GP populations in the way predicted by the BBH. However, there are also several persuasive arguments why GP could not support such building blocks. There continues to be a lack of solid evidence for or against the existence and nature of building blocks in GP.

2.3 Forms of GP Schema

For bit-string chromosomes in GAs there is a widely accepted way of representing schemas in terms of strings from {0,1,*}, where * is a "don't care". There is no such agreement on the form of a schema in GP. Koza [15] defined a schema as the programs which contain a specified set of subtrees. O'Reilly and Oppacher [4] defined a schema more generally as the programs which contain a specified set of fragments at specified frequencies. Whigham [7] used partial derivation trees as schemas for his Context-Free-Grammar based Genetic Programming. Rosca and Ballard [16] defined a schema as the programs which contain a specified fragment at the root. Poli and Langdon [5] modified the schema of Rosca and Ballard by making use of the = node, a "don't care" which takes the place of any single node.

3 Fragment Schemas

The algorithm presented in this paper identifies schemas represented by tree-fragments (or *fragments*). We assume that the programs are represented by rooted trees, where the internal nodes are functions and the leaves are terminals (for example features and constants). A subtree at a node n in a program is the connected set of nodes including n and the nodes below n.

A fragment of a program tree is defined as a connected set of nodes from the program tree. A *rooted fragment* of a program tree is defined as a connected set of nodes from the tree that includes the root of the tree. The set of fragments of a set of programs is the same as the set of rooted fragments of the set of all subtrees of the programs. We will use "fragment" exclusively to refer to the rooted fragments of the programs' subtrees.

A *fragment schema* is defined as the set of all programs containing a specified fragment at some point in the program tree. A fragment f is said to *contain* another fragment g if the root of g is the root of f and the nodes of g are all in f. A fragment is said to be a *containing fragment* of the fragments it contains (other than itself), which are said to be its *contained fragments.*

Figure 1 shows a small program tree, its subtrees, and its fragments.

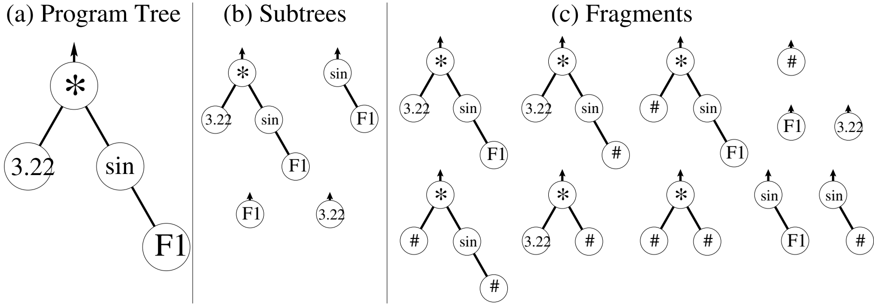


Fig. 1. A program-tree (a) and its component subtrees (b) and fragments (c)

4 Maximal Fragments

This paper introduces a method for performing analysis on the set of all fragments contained in a population of genetic programs. The critical problem that must be addressed is how to represent and how to construct such sets.

For small populations of small programs, naive methods that simply enumerate all fragments are tractable. However, as the sizes or number of the programs increase, the scale of the problem quickly becomes massive, and the naive methods impractical. There are $O(2^{2^d})$ distinct fragments contained in a single, full, binary program tree of depth d, and even a single small program of depth 6 could contain over 10^{11} distinct fragments, too many for a typical modern computer to store or work with.

If a fragment is contained in a subtree (or set of subtrees) then all its contained fragments are also contained in the subtree (or set of subtrees). Therefore, we can use a fragment to represent a set of fragments. The key observation that makes tractable analysis of the fragments in GP populations possible is that the set of fragments contained in a population of genetic programs can be represented by a far smaller set of *maximal fragments*, while still allowing the desired analysis. A fragment f is maximal with respect to a particular set of subtrees S if no containing fragment of f is contained by the same subset of S as the fragment; thus, such a subset of S for any containing fragment of a maximal fragment will always be smaller than the subset for the maximal fragment itself. Given any subset of S, the largest fragment contained in all subtrees in the subset is always maximal with respect to S.

Each maximal fragment f represents a set of fragments; this set includes f and all fragments that are contained in f, that are not also contained in a maximal fragment that is itself contained in f. Therefore, any fragment represented by f must be contained in exactly the same set of subtrees as f. By this mechanism, the set of maximal fragments of a population represents the set of all fragments.

To produce the set of maximal fragments, the method first extracts the set of subtrees from a set of programs. The TripS algorithm, to be described in

the next section, then identifies the maximal fragments rooted at the subtrees. The set of maximal fragments can then be analyzed in various ways to identify properties of the set of all fragments. Section 6 illustrates this analysis process by constructing a histogram of the sizes of all fragments.

5 The TripS Algorithm

TripS [1] is a recursive algorithm that is passed a maximal fragment f to explore and a set of maximal fragments found so far. It will find all containing fragments of f that are maximal fragments, add them to the maximal fragments found so far, and recurse on the new fragments. The key to its efficiency is that it represents maximal fragments by a set of subtrees, and only constructs the fragments explicitly when needed.

Given a subset S of the set S_0 of all subtrees, there is always one largest rooted maximal fragment (possibly the empty fragment) that is in all the subtrees in S; this fragment can be represented by S. TripS is initially called with the maximal fragment represented by the input set of subtrees S_0: TripS($S_0, \{S_0\}$).

Given a maximal fragment, f, to expand, TripS first constructs the explicit representation of the fragment, considers each possible way of extending the fragment with a single node, and then constructs the set of subtrees containing the extended fragment. This set must be a proper subset of the set representing f because f is maximal. This set must also correspond to a maximal fragment. It therefore adds this fragment to the list of fragments found so far, and recurses (unless the fragment has already been found). The TripS algorithm is given in figure 2 as pseudocode.

```
TripS(set of subtrees S, set of sets of subtrees MaxFragSets)
  Fragment f = constructExplicitFragment(S)
  for each single-step extension to f and resulting fragment f′
    S′ = containingSubtrees(S, f′)
    *
    if S′ ∉ MaxFragSets
      add S′ to MaxFragSets
      MaxFragSets =TripS(S′,MaxFragSets)
  Return MaxFragSets
```

Fig. 2. The TripS algorithm: Finding maximal fragments given a set of subtrees

Note, the actual algorithm also stores the Directed Acyclic Graph (DAG) of contained fragments and containing fragments. An edge between S (contained fragment) and S' (containing fragment) is added to this DAG at the point marked * in the algorithm.

[1] From "Subtree-Set-Splitter".

6 Analysis of GP Using TripS

Because the set of maximal fragments is a concise representation of the set of all fragments, it can be tricky to perform analyses on the set of all fragments without explicitly reconstructing all the fragments. Different analyses may require different techniques. Here, we illustrate how statistics on the set of all fragments can be obtained directly from the compact representation.

The algorithm in figure 3 finds a histogram of the number of all fragments of each size that are represented by a set of maximal fragments. The algorithm works as follows: for each maximal fragment f in the set, ContainedFragments (see appendix A) finds a histogram by size of all fragments contained in f. The function then subtracts from this histogram the counts of fragments which are represented by maximal fragments that are themselves contained fragments of f. By examining fragments of increasing size, the algorithm ensures that the histograms for these maximal contained fragments were previously calculated.

```
RepresentedFragments(set of fragments MaxFrags)
  For each f ∈ MaxFrags by topological sort from small fragments to large
    f.repf =ContainedFragments(f)
    For each fragment f′ ∈ MaxFrags that is a contained fragment of f
      For each i: f.repf[i] = f.repf[i] − f′.repf[i]
```

Fig. 3. Finding counts of fragments represented by a set of maximal fragments

Since all the fragments counted in the histogram for a maximal fragment are contained in exactly the same set of subtrees as the maximal fragment, the histograms allow us to group the set of all fragments by size and by the sets of containing subtrees.

6.1 Experimental Setup

We used the algorithm to investigate the behaviour of fragments in runs of evolution for a typical binary classification problem (`breast-cancer-wisconsin` dataset [17] obtained from the University of Wisconsin Hospitals, Madison by Dr. William H. Wolberg. This dataset has 9 features and 699 patterns). The functions used were the four basic mathematical operations ($+, -, \times,$ protected$\div$) each with a fixed arity of two. The terminals used were numeric terminals (normally distributed with $\mu = 0, \sigma = 1$), and feature terminals drawn from the dataset after scaling to [-1,1]. The dataset was divided equally into training, test and validation sets. The population size was kept constant through evolution, with the initial population being generated using the *ramped half-and-half* method. The genetic operators used were reproduction (10%), subtree mutation (30%), and subtree crossover (60%).

6.2 Experiment 1: Numbers of Fragments Sampled

The first set of experiments focused on the numbers of the different sizes of fragments that exist in a population of programs, and how these numbers change over a run of evolution. The population was made up of 300 programs limited to between three and seven nodes deep.

Figure 4(a) displays a histogram of the numbers of fragments, according to size, contained in each generation of a run of evolution. For a particular size and generation, the height of the graph gives the number of distinct fragments of that size, contained in that generation. (The height is plotted on a log scale.) The dotted line seen along the top of the curve at each generation shows how the most numerous size of fragment changes through evolution.

Figure 4(b) displays the same graph seen from above, showing clearly changes from generation to generation of the maximum size of fragment and (dotted line) the most numerous size of fragment.

Figure 4(c) displays the same graph seen from the front, showing the number of fragments of the most numerous size, for each generation.

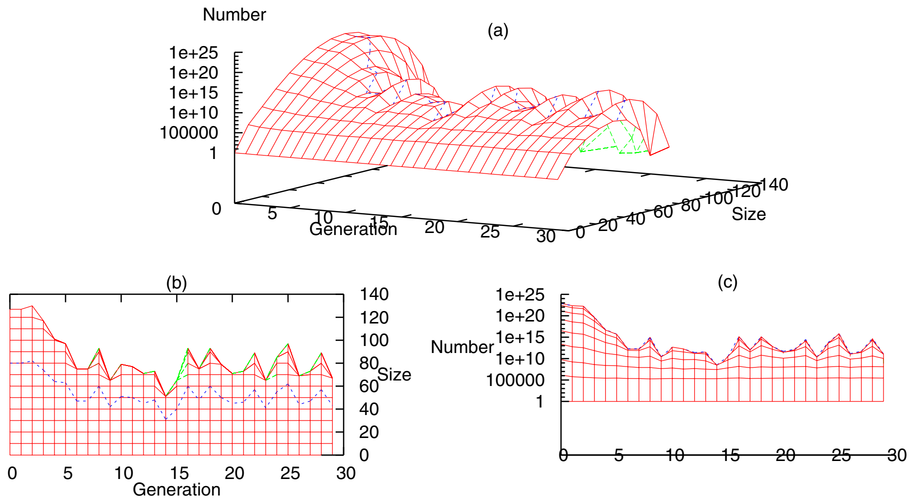


Fig. 4. Histogram of fragments. (a) Height of graph for a particular size and generation shows the number of distinct fragments of the size in the generation (log scale). (b) shows the same graph from above. (c) shows the same graph from the front. The dotted line shows the most frequent size in each generation.

The distribution of each generation's fragments is as expected - few very small or very large fragments, and great numbers (at times over 10^{22}) of middle-sized fragments. It is perhaps of more interest to see how the distribution changes through evolution. We see that the most frequent size of fragments changes from generation to generation, and the total number of fragments sampled by the population varies drastically; for example, the fourteenth generation contained

less than $10^{-12}\%$ as many fragments as the initial population, dropping from 8×10^{22} to 3×10^{8} fragments. In a single step, the number of fragments in the population may rise or fall many orders of magnitude. This is due to the huge increase in the number of fragments a program contains with an increase in the program size; a single large program may contain many times the number of fragments as a population of small programs.

6.3 Experiment 2: Rise in Popularity of Most Frequent Fragments

The second set of experiments focused on the fragments that were most frequent in late stages of the run. The population was made up of 100 programs limited to between three and seven nodes deep.

For each size from three to ten, the most frequent fragments of the size in the sixty-eighth generation were identified. These fragments were then identified in previous generations; The graphs in figure 5 show their frequencies in each generation up to the sixty-eighth. Figure 5 displays two different views of the same graph. For a particular size and generation, the height of the graph gives the number of instances in the generation of the fragment that was the most frequent fragment of the size in the final generation.

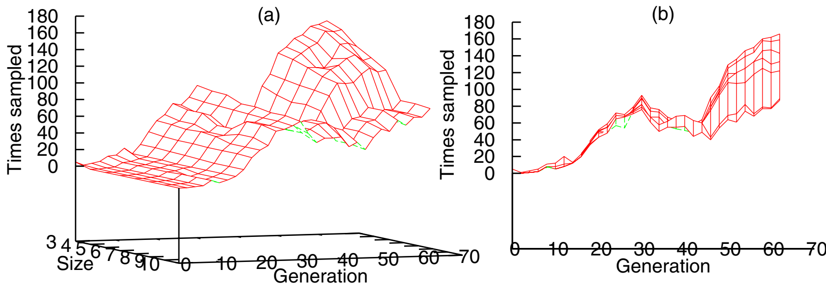


Fig. 5. Retrospective frequency trend of popular fragments showing two views of the same graph: The height of the graph for a particular size and generation reflects the number of subtrees in the generation that contain the the most frequent fragment of the size in the last generation

We see from the graph that the most popular fragments in the final generation had already emerged in the sixth generation. Their sampling frequency then increased, independently of size, to peak in the twenty-ninth generation. After falling briefly, the fragments' popularity increased again, but this time with the smaller fragments becoming significantly more popular than the larger ones. It may be that the evolution went through two phases: the first being an increase in the frequency of these good fragments, and the second being the manipulation of the (now common) fragments in the converged population.

While the analysis here is very preliminary, we hope that this kind of analysis will shed light on the accuracy of predictions made by the GP building block hypothesis, and GP schema theory.

6.4 Experiment 3: Number of Maximal Fragments

The third set of experiments focused on how the number of maximal fragments changes over the course of a run, and from run to run. Evolution was run to generation 100 twenty-five times, with a population size of 100. In each generation in each run the TRIPS algorithm was used to find the number of maximal fragments in the population. The distribution of the numbers at each generation for the twenty-five runs are plotted in figure 6. The figure shows the number

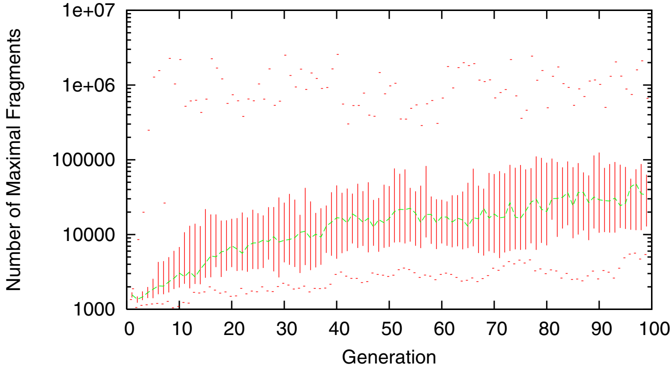


Fig. 6. Distribution of the number of maximal fragments, over 25 evolutions, for each generation up to 99. Shown are the 0th, 25th, 50th, 75th and 100th percentiles. The population size is 100 programs. The vertical axis is on a log scale.

of maximal fragments on the vertical axis (log scale), and generation number on the horizontal axis. The figure shows clearly that the number of maximal fragments varies widely from run to run. Interestingly, while the median number of maximal fragments increases with generation, the maximum number of maximal fragments fluctuates widely from generation to generation, but does not increase on average. It should be noted that the algorithm failed to finish, and was stopped at 10 million maximal fragments, on a single problematic population. The next greatest number of maximal fragments from the 2500 populations was 2.6 million.

By generation 99, the median number of maximal fragments has risen to a modest 33 thousand. The median number of maximal fragments rises steadily with generation, but does not do so at an exponential rate.

TRIPS **Algorithm Complexity.** This section uses the results from the previously described experiment for simple empirical analysis of the complexity of the TRIPS algorithm. From the algorithm, it is immediately clear that there is exactly one call to the TRIPS function for each maximal fragment. During this experiment we also measured how often the inner loop of the algorithm was

Table 1. Distribution over 2500 populations, each with 100 programs, of: number of maximal fragments, number of inner loop iterations, ratio of inner loop iterations to TripS function calls, and the time taken to run the TripS algorithm

	Percentile						
	0%	10%	25%	50%	75%	90%	100%
Num Maximal Fragments	1,048	2,181	4,578	12,131	43,243	141,289	>10,000,000
Runs of inner loop	3,733	17,194	58,997	245,198	1,046,222	4,016,326	94,959,119
Runs of inner loop per call	3.54	8.04	12.95	19.42	28.85	25.05	41.01
Time for TripS (ms)	10	50	150	710	3480	14300	—

iterated (marked with a star in figure 2). Table 1 shows this and other data gained from these experiments.

At times the inner loop is called forty times for each fragment made. Improvements to the algorithm will reduce this ratio significantly. The median time to find all maximal fragments for populations of 100 programs was under one second (on a standard 2.8GHz Pentium IV PC), however, occasionally this increases substantially.

In summary, the algorithm's complexity is closely related to the number of maximal fragments in the population it is run on. Where this number is small, as is found in 95% of the populations tried, the algorithm is efficient. At times, however, the number of maximal fragments is very large, and the algorithm can take a long time to finish.

7 Conclusions

A population of one hundred programs, each 7 deep can easily have trillions of fragment schemas of interest, making impractical any approach that constructs or identifies all fragments explicitly. For this reason, previous attempts to empirically analyze schemas in GP populations have been restricted to very small populations, small programs or restrictive forms of schema.

We have developed the TripS algorithm for finding a set of *maximal fragments* for an input set of programs, which compactly represents the set of all fragments contained in the programs. We have shown the use of another algorithm to derive statistics on the number and sizes of the fragments represented by each maximal fragment, without having to construct all fragments explicitly. The method allows analysis of all fragment schemas in relatively large (300 program) populations of relatively large (over 150 node) tree-based genetic programs.

We present two simple analyses made possible by the method. In the first, a histogram of fragments by size and generation, we see that the number of fragments in the population undergoes massive changes from generation to generation. In the second analysis presented, a plot showing the increasing frequencies

of selected good fragments of different sizes, we see that as well as increasing, the frequency of good fragments can also decrease.

In a third set of experiments we analyzed further the complexity of the algorithm. The results suggest that the number of maximal fragments tends to rise during evolution, but for typical runs remains small. Some rare populations however produce very large numbers of maximal fragments. At all times in the number of maximal fragments remains tiny compared to the total number of fragments.

This method enables many analyses of fragment schemas in real-sized GP populations over and above the two simple examples presented here. Some areas to be looked at in future research include:

– Fitness correlations between programs that share fragments;
– How fragments of different sizes and estimated fitnesses propagate through evolution, specifically looking for and studying building blocks;
– Accuracy of fragment schema theory in practice;
– Origins of the fragments that make up the best-in-run program;
– Further analysis of the algorithm complexity and numbers of maximal fragments;
– Analysis of rooted fragments; and
– Modification of the algorithm to GP hyperschemas and GA schemas.

References

1. Holland, J. *Adaption in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, 1975.
2. Goldberg, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.
3. Altenberg, L. The Schema Theorem and Price's Theorem. In *Foundations of Genetic Algorithms 3* (Estes Park, Colorado, USA, 31 July–2 Aug. 1994), L. D. Whitley and M. D. Vose, Eds., Morgan Kaufmann, pp. 23–49. Published 1995.
4. O'Reilly, U. M., and Oppacher, F. The troubling aspects of a building block hypothesis for genetic programming. Working Paper 94-02-001, Santa Fe Institute, 1399 Hyde Park Road Santa Fe, New Mexico 87501-8943 USA, 1992.
5. Poli, R., and Langdon, W. B. A new schema theory for genetic programming with one-point crossover and point mutation. Technical Report CSRP-97-3, School of Computer Science, The University of Birmingham, B15 2TT, UK, Jan. 1997. Presented at GP-97.
6. Rosca, J. P. Analysis of complexity drift in genetic programming. In *Genetic Programming 1997: Proceedings of the Second Annual Conference* (Stanford University, CA, USA, 13-16 July 1997), J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, Eds., Morgan Kaufmann, pp. 286–294.
7. Whigham, P. A. A schema theorem for context-free grammars. In *1995 IEEE Conference on Evolutionary Computation* (Perth, Australia, 29 Nov. - 1 Dec. 1995), vol. 1, IEEE Press, pp. 178–181.

8. Poli, R., and Langdon, W. B. An experimental analysis of schema creation, propagation and disruption in genetic programming. Technical Report CSRP-97-8, University of Birmingham, School of Computer Science, Feb. 1997. Presented at ICGA-97.
9. Langdon, W. B., and Banzhaf, W. Repeated patterns in tree genetic programming. In *Proceedings of the 8th European Conference on Genetic Programming*, M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, Eds., vol. 3447 of *Lecture Notes in Computer Science*, Springer, pp. 190–202.
10. Wilson, G. C., and Heywood, M. I. Context-based repeated sequences in linear genetic programming. In *Proceedings of the 8th European Conference on Genetic Programming* (Lausanne, Switzerland, 30 Mar. - 1 Apr. 2005), M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, Eds., vol. 3447 of *Lecture Notes in Computer Science*, Springer, pp. 240–249.
11. Majeed, H. A new approach to evaluate GP schema in context. In *Genetic and Evolutionary Computation Conference (GECCO2005) workshop program* (Washington, D.C., USA, 25-29 June 2005), F. Rothlauf et al., Eds., ACM Press, pp. 378–381.
12. Daida, J. M., Bertram, R. R., Polito 2, J. A., and Stanhope, S. A. Analysis of single-node (building) blocks in genetic programming. In *Advances in Genetic Programming 3*, L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, Eds. MIT Press, Cambridge, MA, USA, June 1999, ch. 10, pp. 217–241.
13. Ryan, C., Majeed, H., and Azad, A. A competitive building block hypothesis. In *Genetic and Evolutionary Computation – GECCO-2004, Part II* (Seattle, WA, USA, 26-30 June 2004), K. Deb et al., Eds., vol. 3103 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 654–665.
14. Sastry, K., O'Reilly, U.-M., Goldberg, D. E., and Hill, D. Building block supply in genetic programming. In *Genetic Programming Theory and Practice*, R. L. Riolo and B. Worzel, Eds. Kluwer, 2003, ch. 9, pp. 137–154.
15. Koza, J. R. Genetic programming: On the programming of computers by means of natural selection. *Statistics and Computing 4*, 2 (June 1994).
16. Rosca, J. P., and Ballard, D. H. Rooted-tree schemata in genetic programming. In *Advances in Genetic Programming 3*, L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, Eds. MIT Press, Cambridge, MA, USA, June 1999, ch. 11, pp. 243–271.
17. Newman, D., Hettich, S., Blake, C., and Merz, C. UCI repository of machine learning databases *http://www.ics.uci.edu/∼mlearn/mlrepository.html*, 1998.
18. Mangasarian, O. L., and Wolberg, W. H. Cancer diagnosis via linear programming. SIAM News, Volume 23, Number 5, September 1990, pp. 1 and 18.

A Algorithm to Find Histogram of Sizes of Contained Fragments

The function ContainedFragments, shown in figure 7, is given a maximal fragment f and finds a histogram of the sizes of all fragments contained in f. In the returned array, $c[i]$ is the count of such fragments of size i.

The function recurses once for each node in the fragment.

```
ContainedFragments(fragment f)
  c:Array of counts of fragments of size 0..|f|
  c[0] = 1
  if the root node of f is a function
    For each child node n' of the root node of f
      f' = the fragment of nodes from f including n' and nodes below n'
      c'=ContainedFragments(f')
      For i = |c'| + |c|to 0, step -1: c[i] = Σ_{j=0}^{i} c[j] * c'[i − j]
  For each i in c: c[i + 1] = c[i]
  c[0] = 1
  Return c
```

Fig. 7. Finding the number and sizes of fragments contained in a given fragment

Empirical Comparison of Evolutionary Representations of the Inverse Problem for Iterated Function Systems

Anargyros Sarafopoulos[1] and Bernard Buxton[2]

[1] National Centre for Computer Animation, Bournemouth University, Poole, Dorset, UK
asarafop@bournemouth.ac.uk
[2] Department of Computer Science, University College London, London, UK
B.Buxton@cs.ucl.ac.uk

Abstract. In this paper we present an empirical comparison between evolutionary representations for the resolution of the inverse problem for iterated function systems (IFS). We introduce a class of problem instances that can be used for the comparison of the inverse IFS problem as well as a novel technique that aids exploratory analysis of experiment data. Our comparison suggests that representations that exploit problem specific information, apart from quality/fitness feedback, perform better for the resolution of the inverse problem for IFS.

1 Introduction

The inverse problem for IFS is an open problem that has applications to Fractal Image Compression, Modelling, Computer Graphics, Shape Representation, as well as applications in Art and Design. Several publications exist that employ EAs for the resolution of the inverse problem for IFSs [9], [8, 11], [4, 12]. Different types of EA have been used for the resolution of the inverse IFS problem. In each case different representations, fitness functions, population structures, and selection schemes were adopted. Different types of IFSs, linear and non-linear, were also tried out. The variety of EA methods applied to the resolution of the inverse problem has produced some promising results, however it is difficult to compare and contrast the efficacy and effectiveness of different approaches. This is made especially hard as there is no set of *test* or *training* shapes consistently used throughout the literature. In most cases experiments employed differing amounts of processing power over different sets of images or shapes. In most empirical studies a small number of shapes (sometimes only one or two shapes) were used making it difficult to arrive at general conclusions or allow for comparisons.

Here we concentrate on the comparison of evolutionary representations for the resolution of the inverse IFS problem. This is achieved by selecting a class of fractal images that are representative of the type of shapes often modelled using IFS. We use this set of "classic" IFS fractals as a benchmark for the effectiveness of various techniques/representations and test these under the same processing power and population structure(s).

Presentation of mean performance graphs and frequency distribution graphs of best of run individuals, as well as parametric tests such as ANOVA or z/t-tests [6] for the comparison of mean performance provide empirical evidence as to which representation performs best. Such analysis is very useful, however more often than not it

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 68–77, 2007.
© Springer-Verlag Berlin Heidelberg 2007

does not provide good indications as to why performance varies between representations/techniques. In order to answer this question we visualise the resulting data offline. This is achieved by drawing the best of generation individuals for each run, which in turn, allows for visual analysis of the models generated by the EA runs. The idea is that we don't simply study the best solutions evolved, we also study the series of mutations or the differences between best individuals across generations. Keeping track of the variation between best of generation individuals in a run provides clues as to the "strategy" or the type of evolutionary paths selected by different representations. In biology, especially in the study of development [3], visual qualitatively analysis of mutations that occur in nature is a standard procedure, here it is used for the analysis of the behaviour of the EA.

2 Iterated Function Systems

Iterated functions systems (IFS) provide an elegant mathematical description and a method for generating a large variety of fractal shapes and models. IFS were originally conceived by Hutchinson [5], and made popular by Barnsley [2] who applied IFS to computer graphics and image compression.

In this paper we work with affine IFS. An affine IFS consists of a finite set of n contractive affine transformations w. A contractive transformation w is a transformation that when applied onto a *shape* will scale that *shape* down. That is, if it is applied repeatedly on a *shape* α, with $\alpha_i = w(\alpha_{i-1})$, then α will be transformed to a single point in space. This point is referred to as the *fixed point* of the transformation w. The *fixed point* of w is invariant, that is, if w it is applied onto its *fixed point* then it will transform the *fixed point* onto itself.

To describe an IFS we define the function $W(\alpha) = \bigcup_{i=1}^{n} w_i(\alpha)$. The result of W is often referred to as a *collage*, that is, W transforms *shape* α to a union or *collage* of scaled down versions of α. It can be shown that, given a set of n contractive transformations w, W is also contractive [7]. That is, if W is applied repeatedly onto a *shape* α, with $\alpha_i = W(\alpha_{i-1})$, then it will transform α onto a fixed *shape*. This *shape* is referred to as the *fixed point* of the transformation W.

Given an arbitrary shape α, we want to find the function W whose *fixed point* is α. This is referred to as the inverse or inference problem for IFS. One way of attacking the problem is using Barnsley's [2] collage theorem, which states that in order to find W we need to find a *collage* of scaled down versions of α whose union is α. For detailed description of IFS fractals and fractal image compression techniques look at [2, 7].

3 The Training Set

In order to test the effectiveness of algorithms for the resolution of the inverse IFS problem we select eight (8) well known affine IFS attractors. The attractors are selected so they are made of self-similar shapes of various degrees of complexity, different types of symmetry, with shapes bound by convex and concave polygons, and have different

topologies, i.e. shapes that contain holes and gaps and shapes that don't. The images in the training set fall into (4) four pairs of similar looking shapes:

- The *square-triangle pair*: contains a square shape and the shape of an equilateral triangle. Both figures are simple polygonal compact shapes with axial symmetry along the vertical. For these shapes we require transformations composed only of translation and scaling in order to form solutions to the inverse problem.
- The *non-compact pair*: contains the well known fractal shape of the Sierpinski Triangle and the shape of a crystal-like figure also known as Sierpinski's Pentagons. Both shapes demonstrate axial symmetry along the vertical, and require transformations composed only of translation and scaling in order to form solutions to the inverse problem. The topology of the *non-compact pair* contains many gaps and holes.
- The *leaf pair*: contains the well known fractal shape of Barnsley's Fern and a maple leaf. Both shapes do not demonstrate strict axial symmetry. Both shapes require transformations that contain a composition of scale, rotation, skew, and translation, as well as singular transformations.
- The *branching pair*: contains shapes that are branching recursive structures. Such shapes are well known and well studied fractals. They are made, typically, of many lines or elongated rectangular components. They require transformations composed of translation, rotation, and scaling for the resolution of the inverse problem, and do not demonstrate strict axial symmetry.

All of the above shapes, which we refer to as the *training set*, have well known affine IFS codes, here their attractors lie within $[-0.5, +0.5]^2$ the unit square centred at the origin. Each member of the training set is a 128×128 black and white bitmap that approximates the equivalent IFS attractor. The *training set* is depicted in the rightmost part of Figure 1.

4 Representations of the Inverse Problem for IFS

We introduce three representations of the inverse problem, which we apply and evaluate throughout the rest of the paper. Each representation considers the inverse problem from a different perspective, or emphasises a different aspect of the problem.

- The first representation is based on a combinatorial approach to the resolution of the inverse problem. It derives from the fact that the most general linear transformation can be generated by composition of a *scaling* with a *rotation* and a *skew*. Equivalently any transformation can be decomposed into three steps: scaling, skewing, and rotating. It is possible therefore to use a small set of primitive maps in order to generate by composition any workable mapping. This allows hierarchical structure and the composition of affine maps as hierarchical Lisp symbolic expressions (S-expressions). Based on this idea we devise a hierarchical S-expression based representation using GP. For implementation details look at [11].

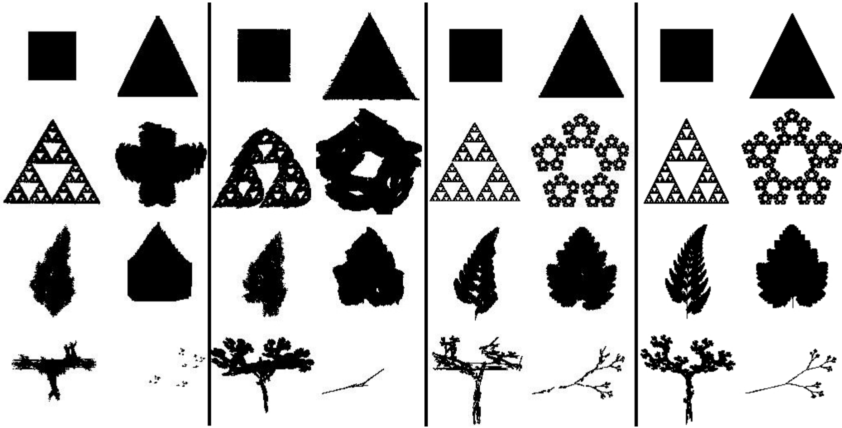


Fig. 1. In the column to the left we display the best individuals evolved using the combinatorial representation, the second column from the left holds the best individuals evolved using the parameter optimisation representation, and the column third from the left illustrates the best individuals evolved using the partitioning representation. Finally, the right hand column depicts the training set.

– The second representation considers a parameter optimisation perspective. This representation is based on an encoding of an IFS as variable-length list of real-valued vectors. This is implemented using GP, where we allow for Evolution Strategies (ES) mutation on GP terminals based on the work on ES by Back [1] and Schwefel [14]. For implementation details look at [12].
– The third representation views the inverse problem as a segmentation or partitioning problem, and exploits problem specific data in order to reduce the search space of the problem. In this case we view the solution of the inverse problem as a partitioning or segmentation task, where (overlapping, or non-overlapping) partitions of a given shape correspond to transformations of that shape as a whole. [1] It is known that in some cases such an approach can reduce the size of the search space for the inverse IFS problem considerably [10]. For implementation details look at [13].

In all three cases the EA algorithm of choice is Genetic Programming which allows for flexible variable length encoding of the problem in hand.

4.1 Population Structure and Fitness Functions

We run experiments for all shapes in the training set, and for each shape we perform runs with 3 different fitness settings. The fitness functions are based on the pixel difference of an in individual's *attractor*, or an individual's *collage* with the appropriate target shape. In the third case the *niches fitness* the population is separated into two niches each of which uses either *collage* or *attractor* fitness. For all shapes and all three fitness settings

[1] That is, we are asked to discover a set of horizontal vertical (HV) partitions for a given shape, each of which is a scaled down transformation of that shape.

we perform 31 independent runs. Each run uses a population of 1000 individuals for 60 generations. That is we process 1.860.000 individuals in the span of 31 independent runs. The population structure is demic using tournament selection, with the number of tournament contestants set to 7. The size of each deme is 3x4 grid squares and the toroid width is 40.

We apply three fitness functions based on the pixel difference of the target shapes and evolved individuals. The *attractor fitness* is calculated as the pixel difference between the target and the *attractor* of evolved IFS. The *collage fitness* is calculated as the pixel difference between the target and the shape generated by the *collage* of the transformations of the target as specified by the evolved IFS. As the attractor is invariant, if we have found the correct solution the target and collage should be identical. Finally we use a *niches fitness* where the population is separated in two niches one using *collage fitness* the other *attractor fitness*.

5 Results

Figure 4 shows the progress of runs for different representations and different target shapes. All representations seem to perform very well for the simple *square-triangle pair*. The partitioning based representation performs the best for the *non-compact pair*, and has a good overall performance for the *leaf pair*. Figure 4 also suggests that the *collage fitness* function provides the most successful solutions, however as we'll see in the following section *collage fitness* can be deceptive and often leads to local minima. Table 1 displays *Student's t* scores for the difference between independent sample means of the performance of representations. We compare mean performance for the shapes in the training set using the three fitness functions described. We test the segmentation/partitioning based representation against the combinatorial and parameter optimisation based representations, each time performing one-tailed tests. The resulting scores indicate that average performance of the partitioning representation is higher for most shapes in the training set. However in the case of *collage fitness* and the comparison between the partitioning and combinatorial representations the results for the tree, and maple leaf appear to be inconsistent with the visual comparison between the best shapes evolved as shown in Figure 1. The same inconsistencies appear between the results depicted in the progress graphs of figure 4, and the visual comparison of best solutions in figure 1. These inconsistencies can be explained by closer inspection of the results. In the following section we point to *deceptive strategies #1 #2* which allow for relatively high fitness scores especially in the case of collage fitness.

6 Visual Analysis

In order to understand why performance is better using the partitioning representation and to resolve inconsistencies we go through the process of visual inspection of the best of generation individuals for each run. We visualise the (sixty) best of generation individuals in a run using a grid of small 128×128 images. The best individual of the first generation is at the top left, and the last individual is placed at bottom right hand corner. The image-grid is read from top to bottom row by row, each row is read from

Table 1. Differences in mean performance between representations for all target shapes. Scores ≥ 1.697 indicate differences with 95% confidence, scores ≥ 2.457 show differences with 99% confidence. We are interested in one-tailed comparison, where Seg_μ stands for the mean performance of the segmentation/partitioning approach, $Comb_\mu$ stands for mean performance of the combinatorial representation, and finally $Optm_\mu$ stands for mean performance of the optimisation based representation.

t-scores Comparison using Attractor Fitness								
	Squar	*Trian*	*Sierp*	*Crys*	*Fern*	*Maple*	*Tree*	*Twig*
$Seg_\mu > Comb_\mu$	8.85	10.03	135.15	23.43	17.68	22.93	5.43	40.82
$Seg_\mu > Optm_\mu$	29.18	15.06	132.42	21.36	11.48	14.98	9.87	50.17
t-scores Comparison using Collage Fitness								
	Squar	*Trian*	*Sierp*	*Crys*	*Fern*	*Maple*	*Tree*	*Twig*
$Seg_\mu > Comb_\mu$	1.07	-4.93	41.43	10.66	3.26	-2.50	-21.27	21.53
$Seg_\mu > Optm_\mu$	10.88	4.09	41.29	11.70	3.34	3.20	-6.15	18.10
t-scores Comparison using Niches Fitness								
	Squar	*Trian*	*Sierp*	*Crys*	*Fern*	*Maple*	*Tree*	*Twig*
$Seg_\mu > Comb_\mu$	80.20	3.69	56.62	20.75	18.04	8.44	-5.84	28.49
$Seg_\mu > Optm_\mu$	16.28	7.16	43.01	16.60	6.58	7.01	0.30	17.50

left to right. In this way, it is possible to inspect all of the best of generation individuals at a glance. Clearly not all individuals are important and many are very similar to one-another, especially in the generations near the end of a run. One could realisticly spend 4 or 5 seconds looking at each grid of images, allowing a quick overview of shapes that hold the phenotypes of sixty related individuals, which means one could inspect all individuals of an experiment in few minutes.

6.1 Deceptive Strategy #1

The grid of images in the left of Figure 2 shows the best of generation individuals of the best run for the maple leaf, using the combinatorial representation. For each best of generation individual one could also view the corresponding set of affine transformations that make up the IFS, see the grid of images on the right of Figure 2. Figure 2 provides a clear example of one of the deceptive "strategies" intended by the EA to resolve the inverse problem, and which typically leads to a local minimum. Even though the best individual using this strategy (shown in Figure 2) achieves an accuracy of 93%. It seems that runs, using collage fitness, for the shapes in the training set and the maple leaf, under the combinatorial representation, fall very often into this trap. That is, GP tries to solve the problem using many scale and mirror transformations, where only one scale transformation (and three transformations using rotation and/or skew) is required for the known optimal solution. We observe that out of 31 runs only 9 runs contain best of generation individuals with a majority of transformations that use rotation and/or skew components in the last 3 generations.

The probability of selecting a scale transformation out of a set made out of *scale, rotation, and skew* at random is $1/3$. A variable X which is the number *scale* transformations appear with majority in the last three generations, out of 31 runs, can be

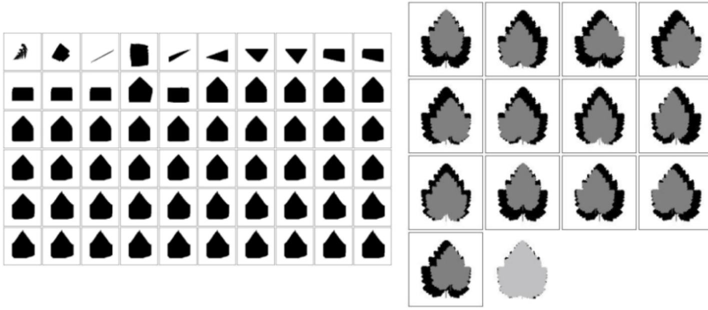


Fig. 2. The grid of images on the left side of the figure contains the best of generation individuals of the best run for the resolution of the inverse problem for the maple leaf using the combinatorial approach and collage fitness. To the right side of the figure the grid of boxes contains the list of affine transformations of the best of run individual for the maple leaf using the combinatorial approach and collage fitness. The last image, on the bottom, drawn in light grey is the collage/union of all the transformations.

modelled by a binomial distribution. Given the null hypothesis H_0 : *a majority of scale transformations appear with probability* $P = 1/3$ at random sampling of runs, and the alternative hypothesis $H_1 : P > 1/3$. We reject the null hypothesis with a significance level of 1%.

6.2 Deceptive Strategy #2

Other deceptive strategies can be discovered this way, Figure 3 provides another example, it depicts the transformations of two individuals, that solve the Sierpinski triangle using the parameter optimisation representation and collage fitness. Each individual is depicted as a grid of images. The individual in the left side is made out of a grid of two transformations, and the individual on the right side is made out of a grid four transformations. Where black pixels draw the target shape and grey pixels draw the

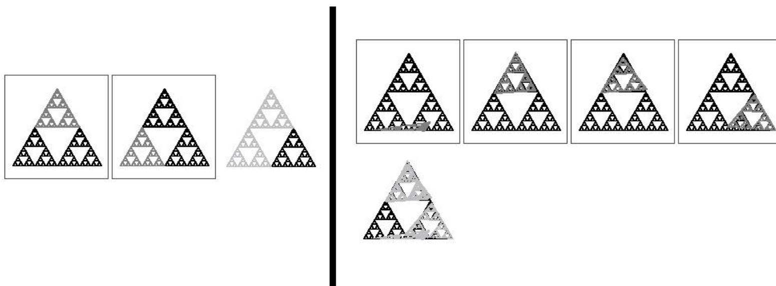


Fig. 3. Two best of generation individuals for the Sierpinski triangle

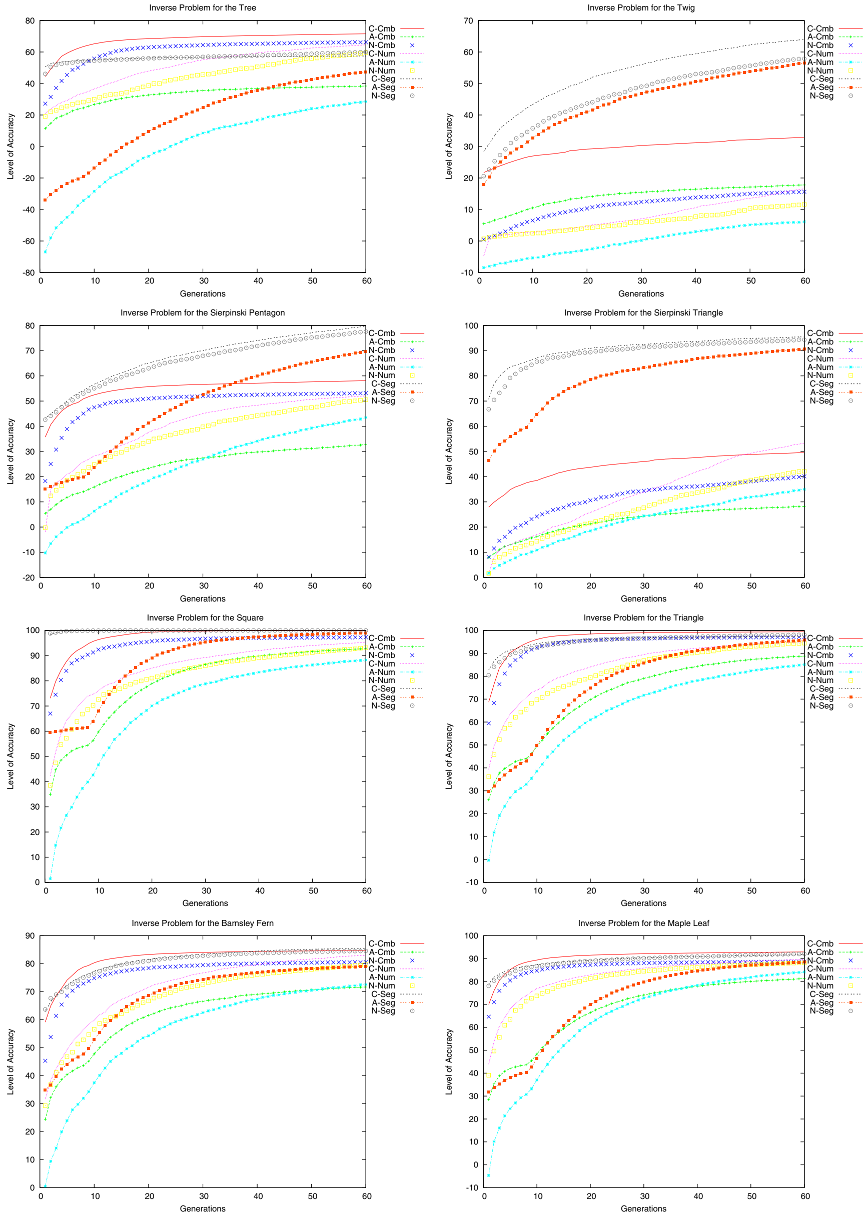


Fig. 4. These graphs show the progress of runs for different representations and different target shapes. They display the average performance of best of generation individuals over 31 runs. The prefix C stands for *collage fitness*, A stands for *attractor fitness*, and N for *niches fitness*. The suffix Cmp refers to the combinatorial representation, Num refers to the numerical optimisation representation, and Seg refers to the segmentation/partitioning representation.

transformed image of the target. The last image on each grid depicts, in light-grey, the collage or union of transformations superimposed onto the target.

Figure 3 highlights a strategy adopted very often. The EA, using the parameter optimisation representation, encourages a search that is quite oblivious to the fact that a solution to the inverse problem requires the whole target shape to be covered. It seems that in many cases a local minimum is reached by covering accurately a small part of the target shape, and at same time avoiding other equally important segments of the target. Runs using collage fitness, for the Sierpinski triangle, fall very often into this trap. The parameter optimisation representation attempts to solve the inverse problem for the Sierpinski triangle by using only part of the actual shape. Out of 31 runs 12 contain best of generation individuals with transformations that cover more than $2/3$ of the target in the last three generations of a run.

The probability of selecting an IFS with a collage of transformations covering (0, 1/3], (1/3, 2/3], or (2/3, 3/3] of a target shape using chance alone should be equal, and the chance of each event occurring equal to 1/3. Given the null hypothesis H_0 : *the collage of transformations of evolved best of generation individuals in the last 3 generations of a run cover more than* $2/3$ *of the target with probability* $P = 1/3$ at random sampling of runs, and an alternative hypothesis $H_1 : P > 1/3$, we find that we can not reject the null hypothesis with an acceptable significance level. In contrast, when using the segmentation based representation to solve the same problem all runs have best of generation individuals whose collages cover $> 2/3$ of the target shape.

7 Conclusions

We have introduced a “training set” of fractal images that can be used as a benchmark for the resolution of the inverse IFS problem. The method for analysis of the results presented in this paper is based on visual inspection of a series of best of generation individuals in EA runs. This method is well suited to the inverse IFS problem, but it could be applied in problems that involve modelling or development of geometric forms. Keeping track of the history of best of generation individuals in a run provides clues as to the “strategy” or the type of evolutionary paths selected by different representations or other EA input parameters. It allows to make simple but important statistical observations that, in this case, indicate the nature of a number of deceptive strategies. Visual inspection of best of generation individuals in EA runs can complement parametric statistical tests and point to reasons for differences in performance of complex input parameters (such as the fitness function) or between variations of EAs.

It seems that the combinatorial and parameter optimisation representations lead to search strategies that often fall into local minima. The segmentation representation provides a reduction of the search space and exploits problem specific information other than quality feedback and therefore performs better in most cases. The approach that appears the weakest of the three in terms of overall results is the parameter optimisation representation. It presents a direct encoding of problem parameters. Of the three representations the other two make obvious use of embedded knowledge about the problem. It is not surprising that embedded knowledge leads to better results. However the segmentation approach apart form the fact that it embeds knowledge in terms of the

application of the *collage theorem*, it also allows the search to proceed by inspecting and using information derived from the target shapes (and therefore of the problem in hand) other than fitness feedback. For example partitions of the shape are made using information directly derived from it, and constrains applied to partitions allow for successful reduction of the search space. It suggests that this indirect feedback allows for better performance.

References

[1] T. Back. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.

[2] M. F. Barnsley and L. P. Hurd. *Fractal Image Compression*. AK Peters, 1993.

[3] E. Coen. *The Art of Genes*. Oxford University Press, 1999.

[4] P. Collet, E. Lutton, F. Raynal, and M. Schoenauer. Polar ifs + parisian genetic programming = efficient ifs inverse problem solving. *Genetic Programming and Evolvable Machines Journal*, 1(4):339–361, October 2000.

[5] J. E. Hutchinson. Fractals and self-similarity. *Indiana University Journal*, 35(5):713–747, 1981.

[6] D. J. Koosis. *Statistics, a Self-Teaching Guide*. John Wiley & Sons, 1997.

[7] N Lu. *Fractal Imaging*. Academic Press, 1997.

[8] E. Lutton, G. Cretin, J. Levy-Vehel, P. Glevarec, and C. Roll. Mixed ifs: resolution of the inverse problem using genetic programming. *Complex Systems*, 9(5):375–398, 1995.

[9] D. J. Nettleton and R. Garigliano. Evolutionary algorithms and the construction of fractals: solution of the inverse problem. *Biosystems*, 33:221–231, 1994.

[10] D. J. Nettleton and R. Garigliano. Reductions in the search space for deriving a fractal set of an arbitrary shape. *Mathematical Imaging and Vision*, 6(4):379–393, 1996.

[11] A. Sarafopoulos. Automatic generation of affine ifs and strongly typed genetic programming. In *Genetic Programming Proceedings of EuroGP1999*, volume 1598 of *LNCS*, pages 149–160. Springer-Verlag, 1999.

[12] A. Sarafopoulos. Evolution of affine transformations and iterated function systems using hierarchical evolution strategy. In *Genetic Programming Proceedings of EuroGP2001*, volume 2038 of *LNCS*, pages 176–191. Springer-Verlag, 2001.

[13] A. Sarafopoulos and B. Buxton. Resolution of the inverse problem for iterated function system using evolutionary algorithms. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 3816–3823. World Congress on Computational Intelligence, IEEE, 2006.

[14] P. H. Schwefel. *Evolution and Optimum Seeking*. John Wiley & Sons, 1995.

Evolution of an Efficient Search Algorithm for the Mate-In-N Problem in Chess

Ami Hauptman and Moshe Sipper

Dept. of Computer Science, Ben-Gurion University, Beer-Sheva, Israel

Abstract. We propose an approach for developing efficient search algorithms through genetic programming. Focusing on the game of chess we evolve entire game-tree search algorithms to solve the *Mate-In-N* problem: find a key move such that even with the best possible counterplays, the opponent cannot avoid being mated in (or before) move N. We show that our evolved search algorithms successfully solve several instances of the Mate-In-N problem, for the hardest ones developing 47% less game-tree nodes than CRAFTY—a state-of-the-art chess engine with a ranking of 2614 points. Improvement is thus not over the basic alpha-beta algorithm, but over a world-class program using all standard enhancements.

1 Introduction

Artificial intelligence for board games is widely based on developing deep, large game trees. In a two-player game, such as chess or go, players move in turn, each trying to win against the opponent according to specific rules. The course of the game may be modeled using a structure known as an adversarial game tree (or simply game tree), in which nodes are positions in the game and edges are moves [10]. The complete game tree for a given game is the tree starting at the initial position (the root) and containing all possible moves (edges) from each position. *Terminal nodes* represent positions where the rules of the game determine whether the result is a win, a draw, or a loss.

When the game tree is too large to be generated completely, only a partial tree (called a search tree) is generated instead. This is accomplished by invoking a *search algorithm*, deciding which nodes are to be developed at any given time, and when to terminate the search (typically at non-terminal nodes due to time constraints) [17]. During the search, some nodes are evaluated by means of an *evaluation function* according to given heuristics. This is done mostly at the leaves of the tree. Furthermore, search can start from any position, and not just the beginning of the game.

In general, there is a tradeoff between *search* and *knowledge*, i.e., the amount of search (development of the game tree) carried out and the amount of knowledge in the leaf-node evaluator. Because deeper search yields better results but takes exponentially more time, various techniques are used to guide the search, typically pruning the game tree. While some techniques are more generic and domain independent, such as alpha-beta search [14] and the use of hash tables

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 78–89, 2007.
© Springer-Verlag Berlin Heidelberg 2007

(i.e., transposition and refutation) [2], other methods rely on domain-specific knowledge. For example, quiescence search [12] relies on examining capture move sequences in chess (and relevant parts of the game tree) more thoroughly than other moves. This is derived from empirical knowledge regarding the importance of capture moves. Theoretically speaking, perfect domain knowledge would render the search futile, as is the case in solved endgames in chess [3]. However, constructing a full knowledge base for difficult games such as chess is still far from attainable.

While state-of-the-art chess engines integrate both search and knowledge the scale is tipped towards generic search enhancements, rather than knowledge-based reasoning [15]. In this paper we evolve a search algorithm, allowing evolution to "balance the scale" between search and knowledge. The *entire search algorithm*, based on building blocks taken from existing methods, is subject to evolution. Some such blocks are representative of queries performed by strong human players, allowing evolution to find ways of correctly integrating them into the search algorithm. This was previously too difficult a task to be done without optimization algorithms, as evidenced by the authors of Deep Blue [5]. Our results show that the number of search-tree nodes required by the evolved search algorithms can be greatly reduced in many cases.

2 Previous Work

Our interest in this paper is in evolving a search algorithm by means of genetic programming. We found little work in the literature on the evolution of search algorithms. Brave [4] compared several genetic-programming methods on a planning problem involving tree search, in which a goal node was to be found in one of the leaves of a full binary tree of a given depth. While this work concluded that genetic programming with recursive automatically defined functions (ADFs) outperforms other methods and scales well, the problems he tackled were specifically tailored, and not real-world problems.

Hong *et al.* applied evolutionary algorithms to game search trees, both for single-player games [10], and for two-player games [11]. Each individual in the population encoded a path in the search tree, and the entire population was evolved to solve single game positions. Their results show considerable improvement over the minimax algorithm, both in speed and accuracy, which seems promising. However, their system required that search trees have the same number of next-moves for all positions. Moreover, they did not tackle real-world games.

Gross *et al.* [7] evolved search for chess players using an alpha-beta algorithm as the kernel of an individual which was enhanced by genetic-programming and evolution-strategies modules. Thus, although the algorithmic skeleton was predetermined, the more "clever" parts of the algorithm (such as move ordering, search cut-off, and node evaluation) were evolved. Results showed a reduction in the number of nodes required by alpha-beta to an astonishing 6 percent. However, since the general framework of the algorithm was determined beforehand, the full power of

evolution was not tapped. Moreover, there is no record of successfully competing against commercial programs, which are known to greatly outperform alpha-beta (with standard enhancements) on specific game-playing tasks.

Previously [9], we evolved chess endgame players using genetic programming, which successfully competed against CRAFTY, a world-class chess program (rated at 2614 points, which places it at the human Grandmaster level), on various endgames. Deeper analysis of the strategies developed [8] revealed several important shortcomings, most of which stemmed from the fact that they used deep knowledge and little search (typically, they developed only *one* level of the search tree). Simply increasing the search depth would not solve the problem, since the evolved programs examine each board very thoroughly, and scanning many boards would increase time requirements prohibitively.

And so we turn to evolution to find an optimal way to overcome this problem: How to add more search at the expense of less knowledgeable (and thus less time-consuming) node evaluators, while attaining better performance. In the experiment described herein *we evolved the search algorithm itself.* While previous work on evolving search used either a scaffolding algorithm [7], or searching in toy problems [4], we present a novel approach of *evolving the entire search algorithm*, based on building blocks taken from existing methods, integrating knowledge in the process, and applying our results to a real-world problem.

We consider all endgames, as opposed to our previous set of experiments [9], in which we only considered a limited subset of endgames. However, an important limit has been imposed: Since efficiently searching the entire game (or endgame) tree is an extremely difficult task, we limited ourselves *for now* to searching only for game termination (or mate positions) of varying tree depths, as explained in the next section.

3 The Mate-In-N Problem

The *Mate-In-N* problem in chess is defined as finding a key move such that even with the best possible counterplays, the opponent cannot avoid being mated in (or before) move N, where N counts only the player's moves and not the opponent's. This implies finding a subtree of forced moves, leading the opponent to defeat in $(2N - 1)$ plies (actually, $2N$ plies, since we need an additional ply to verify a mate). Typically, for such tactical positions (where long forcing move sequences exist), chess engines search much more thoroughly, using far more resources. For example, Deep Blue searches at roughly half the usual speed in such positions [5].

Allegedly, solving the mate problem may be accomplished by performing exhaustive search. However, because deep search is required when N is large, the number of nodes grows exponentially, and a full search is next to impossible. For example, finding a mate-in-5 sequence requires searching 10 or 11 plies, and more than $2 * 10^{10}$ nodes. Of course, advanced chess engines search far less nodes due to state-of-the-art search enhancements, as can be seen in Table 1. Still, the problem remains difficult.

Table 1. Number of nodes required to solve the Mate-in-N problem by CRAFTY—a top machine player—averaged over our test examples. Depth in plies (half-moves) needed is also shown.

Mate-in	1	2	3	4	5
Depth in plies	2	4	6	8	10
Nodes developed	600	7K	50K	138K	1.6M

A basic algorithm for solving the Mate-In-N problem through exhaustive search is shown in Figure 1. First, we check if the search should terminate: successfully, if the given board is indeed a mate; in failure, if the required depth was reached and no mate was found. Then, for each of the player's moves we perform the following check: if, after making the move, *all* the opponent's moves lead (recursively) to Mate-in-$(N-1)$ or better (procedure $CheckOppTurn$), the mating sequence was found, and we return *true*. If not, we iterate on all the player's other moves. If no move meets the condition, we return $false$.

```
MATE-IN-N?(board, depth)

procedure CHECKOPPTURN(board, depth)
 // Check if all opponent's moves lead to Mate-in-(N-1)
 for each oppmove ∈ GETNEXTMOVES(board)
      ⎧ MAKEMOVE(board, oppmove)
      ⎪ result ← MATE-IN-N?(board, depth − 1)
   do ⎨ UNDOMOVE(board, oppmove)
      ⎪ if not result
      ⎩   then return (false)
 return (true)

main
 if ISMATE(board)
   then return (true)
 if depth = 0
   then return (false)
 for each move ∈ GETNEXTMOVES(board)
      ⎧ MAKEMOVE(board, move)
      ⎪ result ← CHECKOPPTURN(board, depth)
   do ⎨ UNDOMOVE(board, move)
      ⎪ if result
      ⎩   then return (true)
 return (false)
```

Fig. 1. A basic algorithm for solving the Mate-In-N problem through exhaustive search

This algorithm has much in common with several algorithms, including alpha-beta search and proof-number (pn) search [1]. However, as no advanced schemas (for example, move-ordering or cutoffs) are employed here, the algorithm becomes infeasible for large values of N.

In the course of our experiments we broke the algorithmic skeleton into its component building blocks, and incorporated them, along with other important elements, into the evolving genetic-programming individuals.

4 Evolving Mate-Solving Algorithms

We evolve our mate-solving search algorithms using Koza-style genetic programming [13]. In genetic programming we evolve a *population* of individual LISP expressions, each comprising *functions* and *terminals*. Since LISP programs may be readily represented as program trees, the functions are internal nodes and the terminals are leaves. (NB: There are two types of tree involved in our work: game search tree and the tree representing the LISP search algorithm.)

Since we wish to develop intelligent—rather than exhaustive—search, our board evaluation requires special care. Human players never develop the entire tree, even when this is possible. For example, Mate-in-1 problems are typically solved by only developing checking moves, and not all possible moves (since non-checking moves are necessarily non-mating moves, there is no point in looking into them). As human players only consider 2–3 boards per second yet still solve deep Mate-in-N problems fast (for example, Grandmasters often find winning combinations more than 10 moves ahead in mere seconds), they rely either on massive pattern recognition or on intelligent pruning, or both [6].

Thus, we evolved our individuals (game search-tree algorithms) accordingly, following these guidelines: 1) Individuals only consider moves adhering to certain conditions (themselves developed by evolution). 2) The amount of lookahead is left to the individual's discretion, with fitness penalties for deep lookahead (to avoid exhaustive search). Thus, we also get evolving lookahead. 3) Development of the game tree is asymmetrical. This helps with computation since we do not need to consider the same aspects for both players' moves. 4) Each node examined during the search is individually considered according to game knowledge, and move sequence may be developed to a different depth.

4.1 Basic Program Architecture

Our individuals receive a chessboard as input, and return a real-valued score in the range $[-1000.0, 1000.0]$, indicating the likelihood of this board leading to a mate (higher is more likely). A representation issue is whether to evolve boards returning scores or moves (allowing to return no move to indicate no mate has been found). An alternative approach might be evolving the individuals as move-ordering modules. However, the approach we took was both more versatile and reduced the overhead of move comparison by the individual—instead of comparing moves by the genetic programming individual, the first level of the search is done by a separate module. An evolved program thus receives as input all possible board configurations reachable from the current position by making one legal move. After all options are considered by the program, the move that received the highest score is selected, and compared to the known solution for fitness purposes (described in Section 4.3).

4.2 Functions and Terminals

We developed most of our terminals and functions by consulting several high-ranking chess players.

Domain-specific functions. These functions are listed in Table 5. Note that domain-specific functions typically examine if a move the player makes adheres to a given condition, which is known to lead to a mate in various positions. If so, this move is made, and evaluation continues. If not, the other child is evaluated. Also, a more generic function, namely $IfMyMoveExistsSuchThat$, was included to incorporate other (possibly unknown) considerations in making moves by the player. All functions undo the moves they make after evaluation of their children is completed. Since some functions are only appropriate in MAX nodes (player's turn), and others in MIN nodes (opponent's turn), some functions in the table were only used at the relevant levels. Other functions, such as $MakeBestMove$, behave differently in MAX nodes and in MIN nodes.

Sometimes functions that consider a player's move are called when it is the opponent's turn. In this case we go immediately to the false condition (without making a move). This solution was simpler than, for example, defining a new set of return types. Some of these functions appear as terminals also, to allow considerations to be made while it is the opponent's turn.

Generic functions appear in Table 4. As in [9], these domain-independent functions were included to allow logic and some numeric calculations.

Chess terminals, some of which were also used used in [9], are shown in Table 6. Here, several mating aspects of the board, of varying complexity levels, are considered. From the number of possible moves for the opponent's king, through checking if the player creates a fork attacking the opponent's king, to one of the most important terminals—$IsMateInOneOrLess$. This terminal is used to allow the player to easily identify very close mates. Of course, repeated applications of this terminal at varying tree depths might have solved our problem but this alternative was not chosen by evolution (as shown below). Material value and material change are considered, to allow the player to make choices involving not losing pieces.

Mate terminals, which were specifically constructed for this experiment, are shown in Table 3. Some of these terminals resemble those from the function set, to allow building different calculations with similar (important) units.

4.3 Fitness

In order to test our individuals and assign fitness values we used a pool of 100 Mate-in-N problems of varying depths (i.e., values of N). The easier 50 problems ($N = 1..3$) were taken from Polgar's Book [16], while those with larger Ns ($N \geq 4$) were taken from various issues of the Israeli Chess Federation Newsletter (`http://www.chess.org.il`). All problems were solved offline by CRAFTY.

Special care was taken to ensure that all of the deeper problems could not be solved trivially (e.g., if there are only a few pieces left on the board, or when the opponent's king can be easily pushed towards the edges). We used CRAFTY's

feature of counting nodes in the game tree and made sure that the amount of search required to solve all problems was close to the average values given in Table 1 (we saved this value for each problem, to use for scoring purposes).

The fitness score was assigned according to an individual (search algorithm's) success in solving a random sample of problems of all depths, taken from the pool (sample size was 5 per N). For each solution, the score was calculated using the formula:

$$fitness = \sum_{i=1}^{s \cdot MaxN} Correctness_i \cdot 2^{N_i} \cdot Boards_i$$

with the following specifications:

- i ,N, and s are the problem instance, the depth, and the sample size, respectively. $MaxN$ is the maximal depth we worked with (currently 5).
- $Correctness_i \in [0, 1]$ represents the percentage of the correctness of the move. If the correct piece was selected, this score is 0.5^d, where d is the distance (in squares) between the correct destination and the chosen destination for the piece. If the correct square was attacked but with the wrong piece, it was 0.1. In the later stages of each run (after more than 75% of the problems were solved by the best individuals), this factor was only 0.0 or 1.0.
- N_i is the depth of the problem. Since for larger Ns, finding the mating move is exponentially more difficult, this factor also increases exponentially.
- $Boards_i$ is the number of boards examined by CRAFTY for this problem, divided by the number examined by the individual[1]. For small Ns, this factor was only used at later stages of evolution.

We used the standard reproduction, crossover, and mutation operators, as in [13]. We experimented with several configurations finally setting on: population size – between 70 and 100, generation count – between 100 and 150, reproduction probability – 0.35, crossover probability – 0.5, and mutation probability – 0.15 (including ERC—Ephemeral Random Constants). The relatively small population size helped to maintain shorter running times, although possibly more runs were needed to attain our results.

5 Results

After each run we extracted the top individual (i.e., the one that obtained the best fitness throughout the run) and tested its performance with a separate problem set (the *test set*), containing 10 problems per each depth, not encountered before. The results from the ten best runs show that all problems up to $N = 4$ were solved completely in most of the runs, and most $N = 5$ problems were also solved.

[1] In order to better control running times, if an individual examined more than 1.5 the boards examined by CRAFTY, the search tree was truncated, although the returned score was still used.

Table 2. Number of search-tree nodes developed to solve the Mate-in-N problem by CRAFTY, compared to the number of nodes required by our best evolved individual from over 20 runs. Values shown are averaged over the test problems. As can be seen, for the hardest problems ($N = 5$) our evolved search algorithm obtains a 47% reduction in developed nodes.

Mate-in	1	2	3	4	5
CRAFTY	600	7K	50K	138K	1.6M
Evolved	600	2k	28k	55K	850k

Table 3. Mate terminal set of an individual program in the population. Opp: opponent, My: player. "Close" means 2 squares or less.

B=IsNextMoveForced()	Is the opponent's next move forced (only 1 possible)?
F=IsNextMoveForcedWithKing()	Opponent must move its king
B=IsPinCloseToKing()	Is an opponent's piece pinned close to the king
F=NumMyPiecesCanCheck()	Number of the player's pieces capable of checking the opponent
B=DidNumAttackingKingIncrease()	Did the number of pieces attacking the opponent's king's area increase after last move?
B=IsPinCloseToKing()	Is an opponent's piece pinned close to the king
B=IsDiscoveredCheck()	Did the last move clear the way for another piece to check?
B=IsDiscoveredProtectedCheck()	Same as above, only the checking piece is also protected

Table 4. Domain-independent function set of an individual program in the population. B: Boolean, F: Float.

F=If3(B, F_1, F_2)	If B is non-zero, return F_1, else return F_2
B=Or2(B_1, B_2)	Return 1 if at least one of B_1, B_2 is non-zero, 0 otherwise
B=Or3(B_1, B_2, B_3)	Return 1 if at least one of B_1, B_2, B_3 is non-zero, 0 otherwise
B=And2(B_1, B_2)	Return 1 only if B_1 and B_2 are non-zero, 0 otherwise
B=And3(B_1, B_2, B_3)	Return 1 only if B_1, B_2, and B_3 are non-zero, 0 otherwise
B=Smaller(B_1, B_2)	Return 1 if B_1 is smaller than B_2, 0 otherwise
B=Not(B)	Return 0 if B is non-zero, 1 otherwise
B=Or2(B_1, B_2)	Return 1 if at least one of B_1, B_2 is non-zero, 0 otherwise

Due to space restrictions we do not present herein a detailed analysis of runs but focus on the most important issue, namely, *the number of search-tree nodes developed by our evolved search algorithms*. As stated above, mates can be found with exhaustive search and little knowledge, but the number of nodes would be

Table 5. Domain-specific function set of an individual program in the population. B: Boolean, F: Float. Note: all move-making functions undo the move when the function terminates.

F=IfMyMoveExistsSuchThat(B, F_1, F_2)	If after making one of my moves B is true, make that move and return F_1, else return F_2
F=IfForAllOpponentMoves(B, F_1, F_2)	If after making each of the opponent's moves B is true, make an opponent's move and return F_1, else return F_2
F=MakeBestMove(F)	Make all moves possible, evaluate the child (F) after each move, and return the maximal (or minimal) value of all evaluations
F=MakeAnyOrAllMovesSuchThat(B, F_1, F_2)	Make all possible moves (in opp turn) or any move (my turn), and remember those for which B was true. Evaluate F_1 after making each of these moves, and return the best result. If no such move exists, return F_2.
F=IfExistsCheckingMove(F_1, F_2)	If a checking move exists, return F_1, else return F_2
F=MyMoveIter(B_1,B_2,F_1,F_2)	Find a player's move for which B_1 is true. Then, develop all opponent's moves, and check if for all, B_2 is true. If so, return F_1, else return F_2
F=IfKingMustMove(F_1, F_2)	If opponent's king must move, make a move, and return F_1, else return F_2
F=IfCaptureCloseToKingMove(F_1, F_2)	If player can capture close to king, make that move and return F_1, else return F_2
F=IfPinCloseToKingMove(F_1, F_2)	If player can pin a piece close to opponent's king, make that move and return F_1, else return F_2
F=IfAttackingKingMove(F_1, F_2)	If player can move a piece into a square attacking the area near opponent's king, make that move and return F_1, else return F_2
F=IfClearingWayMove(F_1, F_2)	If player can move a piece in such a way that another piece can check next turn, return F_1, else return F_2
F=IfSuicideCheck(B,F_1, F_2)	If player can check the opponent's king while losing its own piece and B is true, evaluate F_1, else return F_2

prohibitive. Table 2 presents the number of nodes examined by our best evolved algorithms compared with the number of nodes required by CRAFTY. As can be seen, a reduction of 47% is achieved for the most difficult case ($N = 5$). Note that *improvement is not over the basic alpha-beta algorithm, but over a world-class program using all standard enhancements.*

Table 6. Chess terminal set of an individual program in the population. Opp: opponent.

B=IsCheck()	Is the opponent's king being checked?
F=OppKingProximityToEdges()	The player's king's proximity to the edges of the board
F=NumOppPiecesAttacked()	The number of the opponent's attacked pieces close to its king
B=IsCheckFork()	Is the player creating a fork attacking the opponent's king?
F=NumNotMovesOppKing()	The number of illegal moves for the opponent's king
B=NumNotMovesOppBigger()	Has the number of illegal moves for the opponent's king increased?
B=IsOppKingProtectingPiece()	Is the opponent's king protecting one of its pieces?
F=EvaluateMaterial()	The material value of the board
B=IsMaterialChange()	Was the last move a capture move?
B=IsMateInOneOrLess()	Is the opponent in mate, or can be in the next turn?
B=IsOppKingStuck()	Do all legal moves for the opponent's king advance it closer to the edges?
B=IsOppPiecePinned()	Is one or more of the opponent's pieces pinned?

6 Concluding Remarks

Our results show that the number of search-tree nodes required to find mates may be significantly reduced by evolving a search algorithm with building blocks that provide a-priori knowledge. This is reminiscent of human thinking, since human players survey very few boards (typically 1-2 per second) but apply knowledge far more complex than any artificial evaluation function. On the other hand, even strong human players usually do not find mates as fast as machines (especially in complex positions). Our evolved players are both fast *and* accurate.

GP-trees of our best evolved individuals were quite large, and difficult to analyze. However, from examining the results it is clear that the best individuals' search was efficient, and thus domain-specific functions and terminals play an important role in guiding search. This implies that much "knowledge" was incorporated into stronger individuals, although it would be difficult to quantify it.

The depths (Ns) we dealt with are still relatively small. However, as the notion of evolving the entire search algorithm is new, we expect to achieve better results in the near future. Our most immediate priority is generalizing our results to larger values of N, a task we are currently working on. In the short term we would like to evolve a general Mate-In-N module, which could replace a chess engine's current module, thereby increasing its rating—no mean feat where top-of-the-line engines are concerned!

In the longer term we intend to seek ways of combining the algorithms evolved here into an algorithm playing the entire game. The search algorithms we evolved

may provide a framework for searching generic chess positions (not only finding mates). Learning how to combine this search with the evaluation functions previously developed by [9] may give rise to stronger (evolved) chess players.

Ultimately, our approach could prove useful in every domain in which knowledge is used, with or without search. The genetic programming paradigm still bears great untapped potential in constructing and representing knowledge.

References

1. L. V. Allis, M. van der Meulen, and H. J. van den Herik. Proof-number search. *Artificial Intelligence*, 66:91–124, 1994.
2. D. F. Beal and M. C. Smith. Multiple probes of transposition tables. *ICCA Journal*, 19(4):227–233, December 1996.
3. M. S. Bourzutschky, J. A. Tamplin, and G. McC. Haworth. Chess endgames: 6-man data and strategy. *Theoretical Computer Science*, 349:140–157, December 2005.
4. Scott Brave. Evolving recursive programs for tree search. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 10, pages 203–220. MIT Press, Cambridge, MA, USA, 1996.
5. Murray Campbell, A. Joseph Hoane, Jr., and Feng-Hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1–2):57–83, 2002.
6. C. F. Chabris and E. S. Hearst. Visualization, pattern recognition, and forward search: Effects of playing speed and sight of the position on grandmaster chess errors. *Cognitive Science*, 27:637–648, February 2003.
7. R. Gross, K. Albrecht, W. Kantschik, and W. Banzhaf. Evolving chess playing programs. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 740–747, New York, 9-13 July 2002. Morgan Kaufmann Publishers.
8. A. Hauptman and M. Sipper. Analyzing the intelligence of a genetically programmed chess player. In *Late Breaking Papers at the Genetic and Evolutionary Computation Conference 2005. Washington DC*, June 2005.
9. A. Hauptman and M. Sipper. GP-endchess: Using genetic programming to evolve chess endgame players. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 120–131, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.
10. Tzung-Pei Hong, Ke-Yuan Huang, and Wen-Yang Lin. Adversarial search by evolutionary computation. *Evolutionary Computation*, 9(3):371–385, 2001.
11. Tzung-Pei Hong, Ke-Yuan Huang, and Wen-Yang Lin. Applying genetic algorithms to game search trees. *Soft Comput.*, 6(3-4):277–283, 2002.
12. Hermann Kaindl. Quiescence search in computer chess. *ACM SIGART Bulletin*, (80):124–131, 1982. Reprint in Computer Game-Playing: Theory and Practice, Ellis Horwood, Chichester, England, 1983.
13. John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

14. T. Anthony Marsland and Murray S. Campbell. A survey of enhancements to the alpha-beta algorithm. In *Proceedings of the ACM National Conference*, pages 109–114, November 1981.
15. Monty Newborn. Deep blue's contribution to AI. *Ann. Math. Artif. Intell*, 28(1-4):27–30, 2000.
16. Laszlo Polgar. *Chess : 5334 Problems, Combinations, and Games.* Black Dog and Leventhal Publishers, 1995.
17. S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs , NJ, 1995.

Fast Genetic Programming on GPUs

Simon Harding and Wolfgang Banzhaf

Computer Science Department, Memorial University, Newfoundland
{simonh,banzhaf}@cs.mun.ca
http://www.cs.mun.ca

Abstract. As is typical in evolutionary algorithms, fitness evaluation in GP takes the majority of the computational effort. In this paper we demonstrate the use of the Graphics Processing Unit (GPU) to accelerate the evaluation of individuals. We show that for both binary and floating point based data types, it is possible to get speed increases of several hundred times over a typical CPU implementation. This allows for evaluation of many thousands of fitness cases, and hence should enable more ambitious solutions to be evolved using GP.

Keywords: Genetic programming, Graphics Card Acceleration, Parallel Evaluation.

1 Introduction

It is well known that fitness evaluation is the most time consuming part of the genetic programming (GP) system. This limits the types of problems that may be addressed by GP, as large numbers of fitness cases make GP runs impractical. In some systems it is possible to accelerate the evaluation process using a variety of techniques. In this paper we present a method using the graphics processing unit on the video adapter. We study the evaluation of evolved mathematical expressions and digital circuits, as they are typically used to evaluate the performance of a genetic programming algorithm.

Different approaches have been used in the past for accelerating evaluation. For example, it is possible to co-evolve fitness cases in order to reduce the number of evaluations [1,2]. This, however, adds significant complexity to the algorithm, and does not guarantee an increase in performance under all circumstances. In other applications, one could select the number of fitness cases, e.g.. by stochastic sampling or other methods [3]. Should the system need to be tested against a complete input set, however, this approach would not be suitable. Another method involves compiling the evolved expression to executable code or even using binary code directly [4]. Writing expressions as native code or in a similar vain has many advantages [5]. The compiler or a hand-written algorithm can perform optimisations, e.g. by removing redundant code, which in addition to directly running the expression gives a significant increase in performance. The use of reflection in modern languages such as Java and C♯ provides for the possibility to compile and link code to the currently executing application.

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 90–101, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Under some circumstances it is possible to offload the evaluation to more suitable hardware. When evaluating digital circuits, they can be loaded into a field programmable gate array (FPGA) and then executed on dedicated hardware [6]. This approach can provide large speed increases. However, the downloading of configurations into an FPGA can be a costly overhead. The biggest drawback to this approach is that it requires the use of external hardware, which may have to be specifically developed.

Recently it has become possible to access the processing power of the graphic processing unit (GPU). Modern GPUs are extremely good at performing parallel mathematical operations [7]. However, until recently it was cumbersome to use this resource for general purpose computing. For a general survey on algorithms implemented on GPUs the reader is referred to [8]. For example, discrete wavelet transformations [9], the solution of dense linear systems [10], physics simulations for games, fluid simulators [11], etc., have been shown to be executed faster on GPUs.

In this paper we demonstrate a method for using the GPU as an evaluator for genetic programming expressions, and show that there are considerable speed increases to be gained. Using recent libraries we also show that putting the functions on the GPU to work is relatively painless. As many of these technologies are new, we include web links to sites containing the most recent information on the projects discussed.

Because capable hardware and software are new, there is relatively little previous work on using GPUs for evolutionary computation. For example [12] implements a evolutionary programming algorithm on a GPU, and finds that there is a 5-fold speed increase. Work by [13] expands on this, and evaluates expressions on the GPU. There all the operations are treated as graphics operations, which makes implementation difficult and limits the flexibility of the evaluations. Yu et al [14], on the other hand, implement a Genetic Algorithm on GPUs. Depending on population size, they find a speed up factor of up to 20. Here both the genetic operators and fitness evaluation are performed on the GPU. Ebner et al, use human interaction to evolve aesthetically pleasing shader programs[15]. Here, linear genetic programming structures are compiled into shader programs. The shader programs were then used to render textures on images, which were selected by a user. However, the technique was not extended into more general purpose computation.

To our knowledge, this contribution is the first study of general purpose Genetic Programming, executed on a graphics hardware platform. It makes use of the fact that GP fitness cases are numerous and can be executed in parallel. Provided there is a sufficient number of fitness cases (large datasets), a substantial speedup can be reached.

2 The Architecture of Graphics Processing Units

Graphics processors are specialized stream processors used to render graphics. Typically, the GPU is able to perform graphics manipulations much faster than

a general purpose CPU, as the processor is specifically designed to handle certain primitive operations. Internally, the GPU contains a number of small processors that are used to perform calculations on 3D vertex information and on textures. These processors operate in parallel with each other, and work on different parts of the problem. First the vertex processors calculate the 3D view, then the shader processors paint this model before it is displayed. Programming the GPU is typically done through a virtual machine interface such as OpenGL or DirectX which provide a common interface to the diverse GPUs available thus making development easy. However, DirectX and OpenGL are optimized for graphics processing, hence other APIs are required to use the GPU as a general purpose device. There are many such APIs, and section 3 describes several of the more common ones.

For general purpose computing, we here wish to make use of the parallelism provided by the shader processors, see Figure 1. Each processor can perform multiple floating point operations per clock cycle, meaning that performance is determined by the clock speed and the number of pixel shaders and the width of the pixel shaders. Pixel shaders are programmed to perform a given set of instructions on each pixel in a texture. Depending on the GPU, the number of instructions may be limited. In order to use more than this number of operations, a program needs to be broken down into suitably sized units, which may impact performance. Newer GPUs support unlimited instructions, but some older cards support as few as 64 instructions. GPUs typically use floating point arithmetic, the precision of which is often controllable as less precise representations are faster to compute with. Again, the maximum precision is manufacturer specific, but recent cards provide up to 128-bit precision.

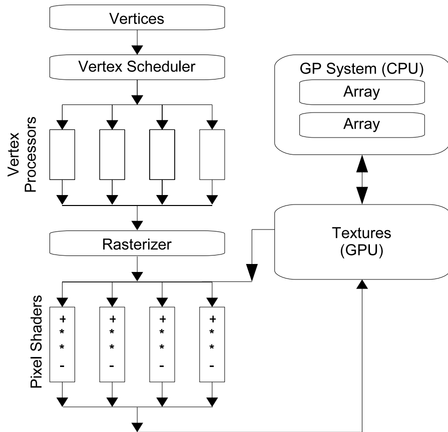


Fig. 1. Arrays, representing the test cases, are converted to textures. These textures are then manipulated (in parallel) by small programs inside each of the pixel shaders. The result is another texture, which can be converted back to a normal array for CPU based processing.

The graphics card used in these experiments is a NVidia GForce 7300 Go, which is a GPU optimized for laptop use. It is underpowered compared to cards available for desktop PCs. Because GPUs are parallel and have very strict processing models, the computational ability of the GPU scales well with the number of pixel shaders. We would therefore expect to see major improvements to the performance of the benchmarks given here if we were to run it on such a GPU. According to [16], "an NVIDIA 7800 GTX 512 is capable of around 200 GFLOPS. ATI's latest X1900 architecture has a claimed performance of 554 GFLOPS". Since it is now possible to place multiple GPUs inside a single PC chassis, this should result in TFLOP performance for numerical processing at low cost.

A further advantage of the GPU is that it uses less power than a typical CPU. Power consumption has become an important consideration in building clusters, since it causes heat generation.

3 Programming a GPU

In this section we provide a brief overview of some of the general purpose computation toolkits for GPUs that are available. This is not an exhaustive list, but is intended to act as a guide to others. More information on these systems can be found at www.gpgpu.org.

SH: Sh is an open source project for accessing the GPU under C++ [17,18]. Many graphics cards are supported, and the system is platform independent. Many low level features can be accessed using Sh, however these require knowledge of the mechanisms used by the shaders. The Sh libraries provide typical matrix and vector manipulations, such as dot products and addition-multiplication operators. In addition to providing general purpose computing, Sh also provides many routines for use in graphics programming. This feature is unique amongst the tools described here, and would be useful in visualisation of results.

Brook: Brook is another way to access the features on the GPU [19]. Brook takes the form of extensions to the C programming language, adding support for GPU specific data types. Applications developed with Brook are compiled using a special C compiler, which generates C++ and Cg code. Cg is a programming language for graphics, that is similar to C. One major advantage of Brook is that it can target either OpenGL or DirectX, and is therefore more platform independent than other tools. However, code must be compiled separately for each target platform. Brook appears to be a very popular choice, and is used for large applications, such as folding@home.

PyGPU: Another recent library allows the access of GPU functionality from the Python language [20]. PyGPU runs as an embedded language inside Python. The work is in its early stages, but results are promising. However it currently lacks the optimization required to make full use of the GPU. It requires a variety of extra packages to be installed into Python, such a NumPy and PyGame (which does not yet support the most recent Python release). Given the rise in popularity

of Python for scientific computing, this implementation should prove useful in the future. Python itself, however, appears to have significant performance issues compared to C++ and JIT languages such as Java or C♯[1].

Accelerator: Recently a .Net assembly called Accelerator was released that provides access to the GPU via the DirectX interface [21]. The system is completely abstracted from the GPU, and presents the end user with only arrays that can be operated on in parallel. Unfortunately, the system is only available for the Windows platform due to its reliance on DirectX. However, the assembly can be used from any .Net programming language.

This tool differs from the previous interfaces in that it uses lazy evaluation. Operations are not performed on the data until the evaluated result is requested. This enables a certain degree of real time optimization, and reduces the computational load on the GPU. In particular, optimisation of common sub expressions, which will reduce the creation of temporary shaders and textures. The movement of data to and from the GPU can also be efficiently optimized, which reduces the impact of the relatively slow transfer of data out of the GPU. The compilation to the shader model occurs at run time, and hence can automatically make use of the different features available on the supported graphics cards.

In this paper we use the Accelerator package. The total time required to reimplement an existing parser tree based GP parser was less than two hours, which we would expect to be considerably less than using any of the other solutions presented here. As with other implementations, Accelerator is based on arrays implemented as textures. The API then allows one to perform parallel operations on the arrays. Conversion to textures, and transfer to the GPU is handled transparently by the API, allowing the developer to concentrate on the implementation of the algorithm. The available function set for operating on parallel arrays is similar to the other APIs. It includes element-wise arithmetic operations, square root, multiply-add, and trigonometric operations. There are also conditional operations and functions for comparing two arrays. The API also provides reduction operators, such as the sum, product, minimum or maximum value in the array. Further functions perform transformations, such as shift and rotate on the elements of the array.

The other systems described here present different variations on these functions, and generally offer functionality that allows different operations to be applied to different parts of the arrays.

4 Parsing a GP Expression

Typically parsing a GP expression involves traversing the expression tree in a bottom-up, breadth first manner. At each node visited the interpreter performs the specified function on the inputs to the node, and outputs the result as the node output. The tree is re-evaluated for every input set. Hence, for 100 test cases the tree would be executed 100 times.

[1] As usual, available benchmarks may not give a fair reflection to real world performance.

Using the GPU we are able to parallelize this process, which means that in effect the tree only has to be parsed once - with the function evaluation performed in parallel. Even without the arithmetic acceleration provided by the GPU, this results in a considerable reduction in computation. Our GP interpreter uses a case statement at the evaluation of each node to determine what function to apply to the input values. If run on the GPU, the tree needs only to be executed once - removing the need for repeatedly accessing the case statement. The use of the GPU is illustrated in Figure 1. The population and genetic algorithm run on the CPU, with evaluations run on the GPU. The CPU converts arrays of test cases to textures on the GPU and loads a shader program into the shader processors. The Accelerator tool kit compiles each individuals GP expression into a shader program. The program is then executed, and the resulting texture is converted back in to an array. The fitness is determined from this output array.

5 Benchmarks

5.1 Configuration

The GP parser used here is written in C♯, and compiled using Visual Studio 2005. All benchmarks were done using the Release build configuration, and were executed on CLR 2.0 on Windows XP. The GPU is an NVidia GeForce 7300 GO with 512Mb video memory. The CPU used is an Intel Centrino T2400 (running at 1.83Ghz), with 1.5Gb of system memory.

In these experiments, GP trees were randomly generated with a given number of nodes. The expressions were evaluated on the CPU and then on the GPU, and each evaluation was timed for evaluation purposes. Timing was performed using calls to Win32 API QueryPerformanceCounter, which returns high precision timings. For each input size/expression length pair, 100 different randomly generated expressions were used, and results were averaged to calculate acceleration factors. Therefore our results show the average number of times the GPU is faster at evaluating a given tree size for a given number of fitness cases. Results less than 1 mean that the CPU was faster at evaluating the expression, values above 1 indicate the GPU performed better.

5.2 Floating Point

In the first experiment, we evaluated random GP trees containing varying numbers of nodes, and exposed them to varying test case sizes. Mathematical functions $+$, $-$, $*$ and $/$ were used. The same expression was tested on the CPU and the GPU, and the speed difference was recorded. Results are shown in Table 1. For small node counts and fitness cases, the CPU performance is superior because of the overhead of mapping the computation to the GPU. For larger problems, however, there is a massive speed increase for GPU execution.

5.3 Binary

The second experiment compares the performance of the GPU at handling boolean expressions. In the CPU version, we use the C♯ boolean type - which is convenient, but not necessarily the most efficient representation. For the GPU, we tested two different approaches, one using the boolean parallel array provided by Accelerator, the other using float. The performance of these two representation is shown in Table 2. It is interesting to note that improvements are not guaranteed. As can be seen in the table, the speed up can decrease as expression size increases. We assume this is due to the way in which large shader programs are handled by either the Accelerator or the GPU. For example, the length of the shader program on the NVIDIA GPU may be limited, and going beyond this length would require repeated passes of the data. This type of behaviour can be seen in many of the results presented here.

We limit the functions in the expressions to AND, OR and NOT, which are supported by the boolean array type. Following some sample code provided with Accelerator, We mimicked boolean behavior using 0.0f as false, and 1.0f as true. For two floats, AND can be viewed as the minimum of the two values. Similarly OR can be viewed as the maximum of the two values. NOT can be performed as a multiply add, where the first stage is to multiply by -1 then add 1.

5.4 Real World Tests

In this experiment, we investigate the speed up on both toy and real world problems, rather than on arbitrary expressions. The GP representation we chose to use here is CGP, but similar results should be obtained from other representations. CGP is fully described in [22]. In the benchmark experiments, the expression lengths were uniform throughout the tests. However, in real GP the length of the expressions vary throughout the run. As the GPU sometimes results in slower performance, we need to verify that on average, there is an advantage.

Regression. We evolved functions that regressed over $x^6 - 2x^4 + x^2$ [23]. We tested the evaluation difference using a number of test cases. In each instance,

Table 1. Results showing the number of times faster evaluating floating point based expressions is on the GPU, compared to CPU implementation. An increase of less than 1 shows that the CPU is more efficient.

	Test Cases					
Expression Length	64	256	1024	4096	16384	65536
10	0.04	0.16	0.6	2.39	8.94	28.34
100	0.4	1.38	5.55	23.03	84.23	271.69
500	1.82	7.04	27.84	101.13	407.34	1349.52
1000	3.47	13.78	52.55	204.35	803.28	2694.97
5000	10.02	26.35	87.46	349.73	1736.3	4642.4
10000	13.01	36.5	157.03	442.23	1678.45	7351.06

Table 2. Results showing the number of times faster evaluating boolean expressions is on the GPU, compared to CPU implementation. An increase of less than 1 shows that the CPU is more efficient. Booleans were implemented as floating point numbers and as booleans. Although faster than the CPU for large input sizes, in general it appears preferential to use the boolean representation. Using floating point representation can provide speed increases, but the results are varied.

Boolean implementation								
	Test Cases							
Expression Length	4	16	64	256	1024	4096	16384	65536
10	0.22	1.04	1.05	2.77	7.79	36.53	84.08	556.40
50	0.44	0.57	1.43	3.02	14.75	58.17	228.13	896.33
100	0.39	0.62	1.17	4.36	14.49	51.51	189.57	969.33
500	0.35	0.43	0.75	2.64	14.11	48.01	256.07	1048.16
1000	0.23	0.39	0.86	3.01	10.79	50.39	162.54	408.73
1500	0.40	0.55	1.15	4.19	13.69	53.49	113.43	848.29
Boolean implementation, using floating point								
	Test Cases							
Expression Length	4	16	64	256	1024	4096	16384	65536
10	0.024	0.028	0.028	0.072	0.282	0.99	3.92	14.66
50	0.035	0.049	0.115	0.311	1.174	4.56	17.72	70.48
100	0.061	0.088	0.201	0.616	2.020	8.78	34.69	132.84
500	0.002	0.003	0.005	0.017	0.064	0.25	0.99	3.50
1000	0.001	0.001	0.003	0.008	0.030	0.12	0.48	1.49
1500	0.000	0.001	0.002	0.005	0.019	0.07	0.29	1.00

Table 3. Results for the regression experiment. The results show the number of times faster evaluating evolved GP expressions is on the GPU, compared to CPU implementation. The maximum expression length is the number of nodes in the CGP graph.

	Test Cases			
Max Expression Length	10	100	1000	2000
10	0.02	0.08	0.7	1.22
100	0.07	0.33	2.79	5.16
1000	0.42	1.71	15.29	87.02
10000	0.4	1.79	16.25	95.37

the test cases were uniformly distributed between -1 to +1. We also changed the maximum length of the CGP graph. Hence, expression lengths could range anywhere from 1 node to the maximum size of the CGP graph. GP was run for 200 generations to allow for convergence. The function set comprised of $+$, $-$, $*$ and $/$. In C♯, division by zero on a float returns "Infinity", which is consistent with the result from the Accelerator library.

Fitness was defined as the sum of the absolute errors of each test case and the output of the expression. This can also be calculated using the GPU. Each individual was evaluated with the CPU, then the GPU and the speed difference

Table 4. Results for the two spirals classification experiment. The results show the number of times faster evaluating evolved GP expressions is on the GPU, compared to CPU implementation. The maximum expression length is the number of nodes in the CGP graph.

	Test Cases			
Max Expression Length	194	388	970	1940
10	0.15	0.23	0.51	1.01
100	0.38	0.67	1.63	3.01
1000	1.77	3.19	9.21	22.7
10000	1.69	3.21	8.94	22.38

Table 5. Results for the protein classifcation experiment. The results show the number of times faster evaluating evolved GP expressions is on the GPU, compared to CPU implementation. The maximum expression length is the number of nodes in the CGP graph.

	Test Cases
Expression Length	2427
10	3.44
100	6.67
1000	11.84
10000	14.21

recorded. Also the outputs from both the GPU and CPU were compared to ensure that they were evaluating the expression in the same manner. We did not find any instances where the two differed.

Table 3 shows results that are consistent with the tests described in previous sections. For smaller input sets and small expressions, it was more efficient to evaluate them on the CPU. However, for the larger test and expression sizes the performance increase was dramatic.

Classification. In this experiment we attempted the classification problem of distinguishing between two spirals, as described in [23]. This problem has two input values (x and y coordinates of a point on a spiral) and has a single output indicating which spiral the point is found. In [23], 194 test cases are used. In these experiments, we extend the number of test cases to 388, 970 and 1940. We also extended the function set to include sin, cos, $\sqrt{x}$, x^y and a comparator. The comparator looks at the first input value to the node, and if it is less than or equal to zero returns the second input, 0 otherwise. The relative speed increases can be seen in Table 4. Again we see that the GPU is superior for larger numbers of test cases, with larger expression sizes.

Classification in Bioinformatics. In this experiment we investigate the behaviour on another classification problem, this time a protein classifier as described in [24]. Here the task is to predict the location of a protein in a cell, from the amino acids in the particular protein. We used the entire dataset as the training set. The

set consisted of 2427 entries, with 19 variables each and 1 output. We investigated the performance gain using several expression lengths, and the results can be seen in Table 5. Here, the large number of test cases used results in considerable improvements in evaluation time, even for small expressions.

6 Conclusions

This paper demonstrates that evaluation of genetic programming expressions can strongly benefit from using the graphics processor to parallelise the evaluations. With new development tools, it is now very easy to leverage the GPU for general purpose computation. However, there are a few caveats. Here we have tested the system using Cartesian GP, however we expect similar advantages with other representations, such as tree and linear GP.

Few clusters are constructed with high performance graphics cards, which will limit the immediate use of these systems. It will require further benchmarking whether low end GPUs found in most PCs today provide a speed advantage. Given the computational benefits and the relatively low costs of fast graphics cards, it is likely that GPU acceleration for numerical applications will become widespread amongst lower priced installations.

Many typical GP problems do not have large sets of fitness cases for two reasons: First, evaluation has always been considered computationally expensive. Second, we currently find it very difficult to evolve solutions to harder problems. With the ability to tackle larger problems in reasonable time we have to also find innovative approaches that let us solve these problems. Traditional GP has difficulty with scaling. For example, the largest evolved multiplier has 1024 fitness cases [25]. In the same time it would take a CPU implementation to evaluate an individual with that many fitness cases, we could test more than 65536 fitness cases on a GPU. This leads to a gap between what we can realistically evaluate, and what we can evolve. The authors of this paper advocate developmental encodings, and using the evaluation approach introduced here we will be able to test this position.

For small sets of fitness cases, the overhead of transferring data to the GPU and for constructing shaders results in a performance decrease. It can be imagined that one would want to determine in practical applications when the advantage of GPU computing kicks in and switch execution to the proper type of hardware. In this contribution, we have just looked at the most trivial way of parallelizing a GP system on GPU hardware. More sophisticated approaches to parallelisation will have to be examined in the future.

References

1. Lasarczyk, C., Dittrich, P., Banzhaf, W.: Dynamic subset selection based on a fitness case topology. Evolutionary Computation **12** (2004) 223–242
2. Gathercole, C., Ross, P.: Dynamic training subset selection for supervised learning in genetic programming. Parallel Problem Solving from Nature III **866** (1994) 312–321

3. Banzhaf, W., Nordin, P., Keller, R., Francone, F.: Genetic Programming - An Introduction. Morgan Kaufmann, San Francisco, CA, USA (1998)
4. Nordin, P., Banzhaf, W., Francone, F.: Efficient Evolution of Machine Code for CISC Architectures using Blocks and Homologous Crossover. In Spector, L., Langdon, W., O'Reilly, U.M., Angeline, P., eds.: Advances in Genetic Programming III, MIT Press, Cambridge, MA, USA (1999) 275–299
5. Brameier, M., Banzhaf, W.: Linear Genetic Programming. Springer, New York, USA (2006)
6. Lau, W.S., Li, G., Lee, K.H., Leung, K.S., Cheang, S.M.: Multi-logic-unit processor: A combinational logic circuit evaluation engine for genetic parallel programming. In: EuroGP. (2005) 167–177
7. Thompson, C., Hahn, S., Oskin, M.: Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis. In: Proceedings of the 35th International Symposium on Microarchitecture, Istanbul, IEEE Computer Society Press (2002) 306 – 317
8. Owens, J., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A., Purcell, T.: A survey of general-purpose computation on graphics hardware. Eurographics 2005, State of the Art Reports (2005) 21–51
9. Wang, J., T. T. Wong, P.A.H., Leung, C.S.: Discrete wavelet transform on gpu. In: Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors. (2004) C–41
10. Galoppo, N., Govindaraju, N., Henson, M., Manocha, D.: Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference (2005) 3– 3
11. Hagen, T.R., Hjelmervik, J.M., Lie, K.A., Natvig, J.R., Henriksen, M.O.: Visual simulation of shallow-water waves. Simulation Modelling Practice and Theory **13** (2005) 716–726
12. Wong, M.L., Wong, T.T., Fok, K.L.: Parallel evolutionary algorithms on graphics processing unit. In: Proceedings of IEEE Congress on Evolutionary Computation 2005 (CEC 2005). Volume 3. (2005) 2286–2293
13. Fok, K.L., Wong, T.T., Wong, M.L.: Evolutionary computing on consumer-level graphics hardware. IEEE Intelligent Systems, to appear (2005)
14. Yu, Q., Chen, C., Pan, Z.: Parallel Genetic Algorithms on Programmable Graphics Hardware. Lecture Notes in Computer Science **3612** (2005) 1051
15. Ebner, M., Reinhardt, M., Albert, J.: Evolution of vertex and pixel shaders. In Keijzer, M., Tettamanzi, A., Collet, P., van Hemert, J., Tomassini, M., eds.: Proceedings of the Eighth European Conference on Genetic Programming (EuroGP 2005), Lausanne, Switzerland, Springer-Verlag (2005) 261–270
16. Wikipedia: Flops — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=FLOPS&oldid=84987291 (2006) [Online; accessed 1-November-2006].
17. RapidMind Inc: Libsh. (http://libsh.org/)
18. LibSh Wiki: Libsh sample code. (http://www.libsh.org/wiki/index.php/Sample_Code)
19. Stanford University Graphics Lab: Brook. (http://graphics.stanford.edu/projects/brookgpu/)
20. Lejdfors, C., Ohlsson, L.: Implementing an embedded gpu language by combining translation and generation. In: SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, New York, NY, USA, ACM Press (2006) 1610–1614

21. Tarditi, D., Puri, S., Oglesby, J.: Msr-tr-2005-184 accelerator: Using data parallelism to program gpus for general-purpose uses. Technical report, Microsoft Research (2006)
22. Miller, J.F., Thomson, P.: Cartesian genetic programming. In et al., R.P., ed.: Proc. of EuroGP 2000. Volume 1802 of LNCS., Springer-Verlag (2000) 121–132
23. Koza, J.: Genetic Programming: On the Programming of Computers by Natural Selection. MIT Press, Cambridge, Massachusetts, USA (1992)
24. Langdon, W.B., Banzhaf, W.: Repeated sequences in linear genetic programming genomes. Complex Systems **15** (2005) 285–306
25. Torresen, J.: Evolving multiplier circuits by training set and training vector partitioning. In: ICES'03:From biology to hardware. Volume 2606. (2003) 228–237

Appendix: Code Examples

To demonstrate the ease of development, we include a small code sample showing the use of MS Accelerator from C♯. The first stage is to make arrays of the data to operate on. In a GP system these may be the fitness cases.

```
float[,] DataA = new float[4096, 4096];
float[,] DataB = new float[4096, 4096];
```

Next, the GPU has to be initialized, and the floating point arrays converted to parallel arrays:

```
ParallelArrays.InitGPU();
FloatParallelArray ParallelDataA =
new DisposableFloatParallelArray(DataA);
FloatParallelArray ParallelDataB =
new DisposableFloatParallelArray(DataB);
```

The parallel arrays are textures inside the GPU memory. Next, the shader program is specified by performing operations on the parallel arrays. However, the computation is not done until requested, as the shader program needs to be compiled, uploaded to the GPU shader processors and executed.

```
FloatParallelArray ParallelResult =
    ParallelArrays.Add(ParallelDataA, ParallelDataB);
```

Finally, we request that the expression is evaluated, and get the result from the GPU. The result is stored as a texture in the GPU, which needs to be converted back into a floating point array that can be used by the CPU.

```
float[,] Result = new float[4096, 4096];
ParallelArrays.ToArray(ParallelResult, out Result);
```

FIFTH™: A Stack Based GP Language for Vector Processing

Kenneth Holladay[1], Kay Robbins[2], and Jeffery von Ronne[2]

[1] Southwest Research Institute, San Antonio, Texas
[2] University of Texas at San Antonio, San Antonio, Texas

Abstract. FIFTH™, a new stack-based genetic programming language, efficiently expresses solutions to a large class of feature recognition problems. This problem class includes mining time-series data, classification of multivariate data, image segmentation, and digital signal processing (DSP). FIFTH is based on FORTH principles. Key features of FIFTH are a single data stack for all data types and support for vectors and matrices as single stack elements. We demonstrate that the language characteristics allow simple and elegant representation of signal processing algorithms while maintaining the rules necessary to automatically evolve stack correct and control flow correct programs. FIFTH supports all essential program architecture constructs such as automatically defined functions, loops, branches, and variable storage. An XML configuration file provides easy selection from a rich set of operators, including domain specific functions such as the Fourier transform (FFT). The fully-distributed FIFTH environment (GPE5) uses CORBA for its underlying process communication.

Keywords: Genetic Programming, vectors, linear GP, GP environment.

1 Introduction

Genetic programming (GP) is a supervised machine learning technique that searches a program space instead of a data space [1]. A crucial factor in the success of a particular GP application is the ability of the underlying GP language (GPL) to efficiently represent solutions in the intended problem domain. For example, Spector [2] observed that PUSH3 was unable to evolve a list sort until the list was stored as an external data structure and list access instructions were added to the GPL. Lack of language expressiveness is particularly acute for large classes of feature extraction problems that arise in many important applications such as time-series data mining and digital signal processing (DSP). Feature extraction and related algorithms are often most compactly expressed using vectors.

A fundamental difficulty in applying traditional GP languages to feature extraction problems is the mismatch between the data handling capacity of the language and the requirements of the applications. When examined from an abstract viewpoint, feature recognition algorithms often contain a series of transformations on vector spaces. Thus, the GP search should explore functions on vector spaces. Unfortunately, common GP languages [2-9], whether stack or tree based, do not treat vectors as single data elements. Even simple vector operations, such as multiplication of a vector by a scalar, require a loop construct with branch and flow control.

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 102–113, 2007.
© Springer-Verlag Berlin Heidelberg 2007

While there have been a few published applications of GP to signal processing problems, the recurring theme has been to circumvent native vector handling rather than to increase the expressiveness of the language. For example, to automatically classify sound and image databases, Teller [10] designed a GP system that used special data access primitives with specific bound constraints rather than using generalized vectors. In applying GP to signal processing algorithms, Sharman [11] also avoided vector functions by introducing delay nodes and processing time series data one point at a time. For pattern recognition, Rizki et al. [12] pre-processed the data to reduce dimensionality (feature extraction) prior to applying GP.

In this paper, we introduce a new Turing complete GP programming language called FIFTH with a native vector handling capability that allows functions on vector spaces to be expressed in a compact form. FIFTH is a stack-based language (patterned after FORTH syntax and early FORTH [13] design considerations) whose linear representation supports robust mutation and crossover operations.

The remainder of this paper is organized as follows. Section 2 presents the details of the FIFTH language and discusses its implementation. Section 3 describes the GPE5™ distributed execution environment. Section 4 shows two example problems, and Section 5 discusses the FIFTH design as well as future work.

2 The FIFTH Language

The key features of FIFTH include a syntax that can be easily manipulated, a single data stack that allows vectors to be retained intact and treated as single data elements, and an internal structure that supports rich program control and new word generation as well as the ability to wrap external functions as language primitives. These ingredients are needed for effective search of the program space for vector problems.

2.1 Syntax, Operators, and the Data Stack

A FIFTH program consists of an input stream of tokens separated by white space. Tokens are either words (operators found in a dictionary) or numbers. All operations take their data from a single data stack and leave their results on this stack in contrast to FORTH and PUSH3, which use separate stacks for each data type. The FIFTH stack holds "containers" with two principal attributes: shape and type. Shape refers to the number of dimensions and the number of elements in each dimension of the data in a container. For data types, FIFTH currently supports `NUMERIC` (integer, real, and complex), `STRING` (UTF-8), and `ADDRESS` (container and word address), as well as arrays in one and two dimensions.

All operators are expected to "do the right thing" with whatever container types they find on the stack. This approach simplifies the syntax for complex vector operations and significantly reduces the number of structural words in the language, effectively reducing the search space and simplifying genetic manipulation.

2.2 Core Vocabulary

Table 1 shows some of the more specialized categories of FIFTH words. The last category in the table, Signal Processing, highlights another strong point of FIFTH.

Using a simple set of wrapper functions, it is easy to incorporate special purpose external libraries such as the GNU Scientific Library. While GP implementations have always tailored their vocabulary to fit the problem, the native vector handling in FIFTH greatly extends the range of available options. This also positively affects the search space by allowing researchers to apply domain specific knowledge.

For example, determining the symbol rate of an encoded signal is an important step in blind radio signal recognition. The phase derivative algorithm (DPDT) [14] can be implemented in several hundred lines of C++ code, but can be expressed compactly in FIFTH as:

```
x ANGLE UNWRAP DIFF MAGNITUDE
FFT MAGNITUDE LENGTH 2.0 / SETLENGTH
LENGTH RAMP 1 + LENGTH Fs SWAP / * 30.0 GT
* LENGTH SWAP MAXINDEX SWAP Fs SWAP / 2.0 / *
```

The token `x` is an input vector representing the signal, and `Fs` is the sampling frequency. The first line develops a feature set from the signal vector (x) by taking the first difference of the unwrapped phase angle. The second line uses a Fourier transform to develop the periodicity of the feature set, while the third and fourth lines find the highest spectral line above 30 Hz and convert its position into a symbol rate using the sampling frequency.

Table 1. Example words in the FIFTH vocabulary

Category	**Word Set**
Stack	`DROP DUP OVER ROT SWAP`
Vector	`MAX MAXINDEX MIN MININDEX ONES ZEROS LENGTH SETLENGTH`
Matrix	`SHAPE HORZCAT VERTCAT TRANSPOSE FLIP`
Definition and Flow	`BEGIN UNTIL IF ELSE THEN WORD ENDWORD VARIABLE CONSTANT`
Signal Processing	`FFT IFFT FFTSHIFT ANGLE UNWRAP dBMAG HAMMING HILBERT MAGNITUDE`

2.3 Formal Aspects and Validation

FIFTH is a type safe language with formal type rules [15]. The type system can provide information to assist the random program generator and genetic manipulator in constructing syntactically and operationally correct programs. Filtering out incorrect programs reduces the search space and improves run-time performance [16].

A selection of important typing rules is shown in Fig. 1. By convention, an arbitrary FIFTH word is represented $\mathcal{W}$, and a sequence of words by $\mathcal{P}$. The type of an individual slot on the stack is expressed with τ, σ, or ρ (with the top of the stack to the right), and sequences of slots are represented with ϕ. The types of FIFTH words are expressed as functions from initial stack typings to new stack typings. The type rules

$$\Delta \vdash \text{ROT} : \tau\ \sigma\ \rho \to \sigma\ \rho\ \tau$$

$$\textbf{COMPOSE}\ \frac{\Delta \vdash \mathcal{W}: \phi_1 \to \phi_2 \quad \Delta \vdash \mathcal{P}: \phi_2 \to \phi_3}{\Delta \vdash \mathcal{W}\ \mathcal{P}: \phi_1 \to \phi_3}$$

$$\textbf{STACK}\ \frac{\Delta \vdash \mathcal{P}: \phi_1 \to \phi_2}{\Delta \vdash \mathcal{P}: \theta \cdot \phi_1 \to \theta \cdot \phi_2}$$

$$\textbf{COMPILE}\ \frac{\Delta, \mathcal{W}: \phi_1 \to \phi_2 \vdash \mathcal{P}_1 : \phi_1 \to \phi_2 \qquad \Delta, \mathcal{W}: \phi_1 \to \phi_2 \vdash \mathcal{P}_2 : \phi_3 \to \phi_4}{\Delta \vdash \text{WORD}\ \mathcal{W}\ \mathcal{P}_1\ \text{ENDWORD}\ \mathcal{P}_2 : \phi_3 \to \phi_4}$$

$$\textbf{TAUT}\ \Delta, \mathcal{W}: \phi_1 \to \phi_2 \vdash \mathcal{W}: \phi_1 \to \phi_2$$

Fig. 1. Example basic type rules for FIFTH

assume an environment of mappings from declared words to their type; these are represented with Δ.

The **COMPOSE** rule allows sequences of FIFTH words to be executed as long as each word's ending stack typing matches the beginning stack typing of the next word. The **STACK** rule says that if a sequence has a valid typing, additional slots can be added to the bottom of the stack (as long as the top of the stack matches the signature) to create new valid typings for that sequence. The **COMPILE** rule allows new FIFTH words to be defined and added to the environment. These can then be used in accordance with the **TAUT** rule.

In addition to formal typing, FIFTH includes support for tracing programs and for comparing the execution of an algorithm in FIFTH with execution in MATLAB through `READMAT` and `WRITEMAT` file operations. These operations read and write MATLAB [17] revision 4 binary files and are used for data I/O as well as debugging. We selected the MATLAB format because it is capable of representing vectors using multiple data types including integer, float, and complex representations. These features allow us to implement all or parts of an algorithm in both languages and exchange data for automatic comparison.

3 The FIFTH Genetic Programming Environment

The Genetic Programming Environment for FIFTH (GPE5) consists of three major components, as illustrated in Fig. 2. GP5 provides random program generation, genetic manipulation, program interpretation and parsing, as well as an interactive FIFTH terminal. DEC5 (Distributed Evaluation Controller) manages the distributed evaluation of programs for each generation. One or more DPI5 (Distributed Program

Interpreter) components are required to run the programs against the problem data sets. Each DPI5 is controlled by the DEC5 through CORBA interfaces.

All components are written in C++ and have been tested on Windows XP, Solaris, and Linux operating systems. Where operating system specific interaction is required, the ACE libraries provide an intermediate layer. CORBA communication between the distributed components is based on the TAO [18] libraries.

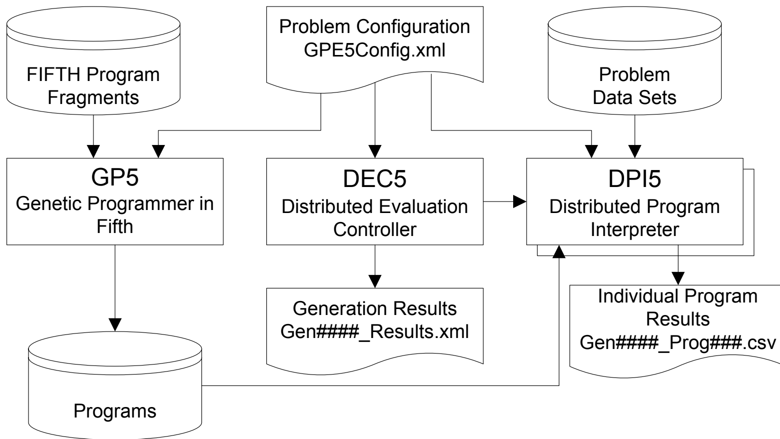


Fig. 2. Block diagram for the FIFTH Genetic Programming Environment (GPE5)

3.1 Random Program Generation

Another way to reduce the search space in GP is to ensure that only syntactically correct programs are allowed into the gene pool. FIFTH was designed so that each word in the dictionary has a stack signature that consists of the number and types of containers expected on the data stack when it is executed and the number and types of containers left on the stack when it finishes.

The random program generator (RPG) uses a two step approach to creating syntactically correct programs with the desired output signature. After first generating a randomly selected fraction of the maximum program size, RPG parses the program to identify errors and fixes them by inserting additional words and terminals.

3.2 Crossover and Mutation in FIFTH

FIFTH supports crossover, mutation, and cloning as standard genetic manipulations, but the infrastructure supports easy addition of any technique that operates on one or two parents. Crossover is performed as follows. First, select two parent programs. In the first parent, select a random number of sequential words in a random location in the program (within configurable constraints) then characterize the input-output effect for the sequence. In the second parent, find all sequences (within configurable constraints) that match the input-output effect then randomly select one. Swapping the two compatible sequences completes the operation. Mutation uses the same initial

sequence selection and characterization, but the sequence is deleted then replaced by a randomly generated sequence that matches the characterization.

A typical problem with mutation and crossover operations in genetic programming is the high probability that the result will destroy the viability of the offspring because the program no longer executes correctly after the operation. Tchernev [19] states that the destructive effects of crossover can be mitigated in FORTH-like GP languages by requiring that the exchanged segments have matching stack signatures. Our implementation of FIFTH uses a stack signature that tracks not only stack size but also branch and loop depth. FIFTH mutation allows a single word or block of words to be replaced by a word or block with a matching stack signature. Crossover operations are similarly constrained. While this choice limits the search space, it increases the probability that the result of evolution will not be destructive.

4 Using GPE5 to Solve a Problem

As with any GP environment, solving a problem using GPE5 requires a few preparatory steps, most of which involve editing the XML configuration file from which all of the components read their control information. The following descriptions are reasonably analogous to the preparatory steps outlined by Koza [20].

4.1 Identify the Terminal Set

In GPE5, there are two classes of terminals: problem data inputs and ephemeral numbers. Problem data inputs may be scalars, vectors, or arrays. For each data set, the values are stored as variables in MATLAB V4 .mat file format. The variable names are in the configuration file for selection during the initial random program generation. An inherent part of this preparatory step is the identification and organization of the data set files. Each data set must be in a separate .mat file. When read into the FIFTH program, each variable in the .mat file becomes a FIFTH CONSTANT. The second category consists of numbers that may be selected during random program generation. In the configuration file, groups of terminal tokens are assigned separate probabilities of selection.

4.2 Identify the Function Set

The function set for GPE5 is divided into three categories controlled by the configuration file. Groups of tokens within each category can be assigned an independent probability of selection. The first category consists of the subset of FIFTH words that are appropriate for the problem under consideration. These may be math operations, logical comparisons, and stack manipulations, as well as domain specific functions such as filters and Fourier transforms.

The second category of functions, unique to GPE5, consists of user user-defined FIFTH program fragments. Each fragment is a valid FIFTH program stored in a file. If a fragment is selected when randomly generating a program, the fragment code is inserted into the program so that it is indistinguishable from the purely random material. In subsequent generations, the fragments are subject to genetic manipulation just like the random parts of the program.

The third category describes the architectural and structural components of a program. These include automatically defined words (WORD, ENDWORD), intermediate storage (VARIABLE, CONSTANT), as well as branch and loop constructs (IF, ELSE, THEN, BEGIN, UNTIL).

4.3 Determine the Fitness Evaluation Strategy

Fitness is a measure of how well a program has learned to predict the output(s) from the input(s). Generally, continuous fitness functions are necessary to achieve good results with GP. While fitness can be any continuous measure, GPE5 uses standardized forms where zero represents a perfect fit. These include error, error squared, and relative error. Although relative error is not often encountered in the GP literature, it works well for problems where the acceptable deviation from the actual output is proportional to the output value.

To avoid overfitting during program evolution, GPE5 uses a different subset of the available data files for each generation. The number of files included in the subset is controlled by a configuration parameter.

4.4 Select the Control Parameters

The GPE5Config.xml file provides a mechanism for experimenting with a number of control parameters including minimum and maximum number of generations, fitness function, fitness target, maximum execution stack depth, and maximum number of words to execute for a single program (to avoid endless loops).

5 Example Problems

5.1 Polynomial Regression

We selected polynomial regression for initial testing of the GPE5 to demonstrate its ability to solve standard GP test problems. Fig. 3 shows an example run for the quadratic equation $y = 0.5x^2 + x/3 - 2.2$. Pertinent configuration parameters were: population size = 1000, minimum program size = 15, maximum program size = 150, parent pool size = 900, and exponential fitness ranking bias = 0.9. The genetic operations included cloning, mutation, and crossover with respective probabilities of 0.05, 0.50 and 0.45. Both mutation and crossover used a uniform length selection of between 1 and 4 program tokens. Table 2 shows the terminal values and functions configured for use by the random program generator.

A typical human generated solution would require about 17 program tokens:

```
1.0 2.0 / x * x * x 3.0 / + 1.0 5.0 / 2.0 + -
```

A longer (43 tokens) but correct general solution was evolved in 33 generations.

```
x x * 1.0  + 1.0  x -2.0  1.0  / - 5.0  - -3.0  -3.0  / -3.0
+ -4.0 - -5.0  / -5.0  + -3.0  / x + -3.0  / OVER - -4.0  / -
- - 2.0  / -4.0  - -5.0  +
```

Table 2. Tokens used by the random program generator for polynomial regression problems

P(selection)	Terminals and Functions in the set
0.20	x (data input)
0.30	1 2 3 4 5 -1 -2 -3 -4 -5
0.40	* + - /
0.10	DUP SWAP OVER ROT

Fig. 3 shows the evolutionary progress. The solid line shows the best fitness at each generation. The dotted line shows the fitness of the best ancestor in each generation for the final program (shown above) that met the fitness termination criteria.

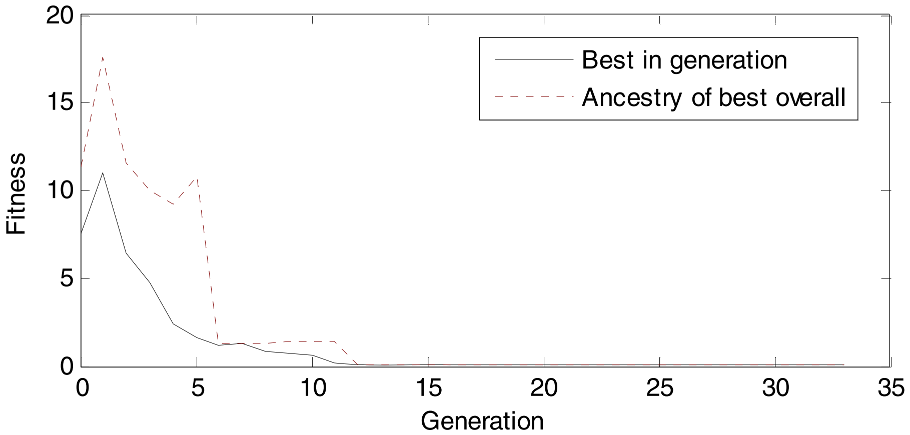


Fig. 3. Best fitness progression for an example run of polynomial regression

5.2 Symbol Rate for Phase Shift Key Modulated Signals

A problem that better illustrates the vector manipulation capabilities of FIFTH and GPE5 is the determination of the symbol rate of phase shift key (PSK) digitally modulated signals. Our previous work [21, 22] compared several common symbol rate algorithms for PSK signals in the High Frequency (HF) radio bands. To prepare a data set for this problem we selected signal property values that are known to work well with the DPDT algorithm presented in section 2.2. Typical values for these properties as well as the specific values used are shown in Table 3. For each unique combination of these properties we generated 10 different signals containing random sequences of symbols. The resulting data set contained 2550 signals. The sample rate was fixed at 8000 complex (In Phase and Quadrature) samples per second, and the signal length was fixed at 16384 samples. Using 1% relative error as the cutoff for a correct symbol rate calculation, the DPDT algorithm achieves 100% correct responses against the entire test data set.

Table 3. Properties of the signals used for the PSK symbol rate example

Property	Typical Values in HF	Values Used
Modulation	FSK, MSK, PSK, DPSK, OQPSK, QAM, ASK	Phase Shift Key
Pulse shape	None, raised cosine (RC), root RC (RRC), Gaussian	Raised Cosine
Excess bandwidth (rolloff)	Limit: 0.00 to <1.00. Typical: 0.10 to 0.35	0.1
Symbol rate	Typical: 10 to 2400 symbols per second	300 to 2400 in increments of 25
Symbol states	2, 4, 8, 16	2, 4, 8
Signal to noise ratio (SNR)	Practical range: 0 to 60 dB	Infinite (no noise)

A simplified form of this problem evolves only the feature extraction stage as described in [21], followed by standard FFT and peak search stages to derive the symbol rate. With the available vector functions in FIFTH, GPE5 quickly produced modified forms of two common algorithms (phase derivative and magnitude squared) that achieved 100% correct responses.

Evolving the entire symbol rate algorithm has been more challenging. We are in the early phases of experimenting with the wide range of configuration options built into GPE5, but initial results are promising. For example, Fig. 4 shows the best fitness progress for a run using a configuration consisting of population size = 4000, minimum program size = 40, maximum program size = 400, parent pool size = 3500, exponential fitness ranking bias = 0.9, with terminals and functions shown in Table 4. The genetic operations included cloning probability = 0.05, mutation probability = 0.55 with uniform length selection between 1 and 10 tokens, and crossover probability = 0.40 with uniform length selections between 1 and 50 tokens.

The best result from this run achieved a performance of almost 70% correct identification against the entire training set.

Table 4. Tokens used by the random program generator for the symbol rate problem

P(selection)	Terminals and Functions in the set
0.10	`x (signal vector)`
0.02	`Fs (sample rate)`
0.13	`1.0 2.0 3.0 4.0 5.0 10.0 -1.0 -2.0`
0.10	`DUP SWAP ROT OVER`
0.05	`LENGTH GT MAX MAXINDEX`
0.25	`ANGLE UNWRAP DIFF MAGNITUDE FFT SQRT REAL IMAGINARY RAMP SETLENGTH MAGSQRD`

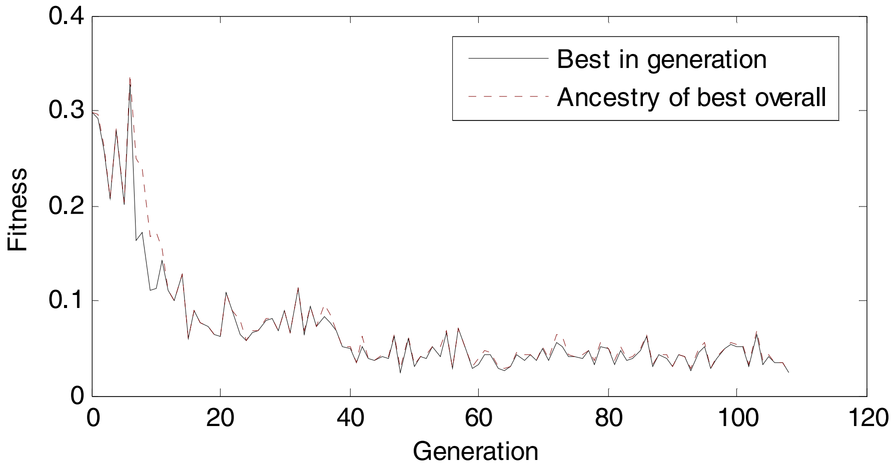


Fig. 4. Best fitness progression for an example run of PSK symbol rate

6 Discussion

The FIFTH programming language was motivated by the need to lift genetic program search to functions on vector spaces by supporting natural, control-free behavior for vector and matrix operations. FIFTH tends to produce mappings from $\mathbb{R}^n \to \mathbb{R}^m$ or in the case of complex values, $\mathbb{C}^n \to \mathbb{C}^m$. These manipulations are natural for a large class of problems including signal processing applications.

Allowing vectors and matrices to be primitive data elements dramatically alters the search space because it eliminates the control operations needed in scalar languages. The probability that such control operations will be damaged during mutation and crossover is high. Even if the control operations are maintained, it is unlikely that intrinsic sizes of vectors will be correctly preserved as successive transformations are performed.

The design philosophy underpinning FIFTH does not mirror the choices that have been made for many linear GPLs and environments. Many authors advocate using function sets and data representation that closely resemble the structure of current CPU hardware, reasoning that this represents a simple structure with significant execution speed advantages. We perceive as a disadvantage to this approach that the resulting programs are almost incomprehensible to humans without significant effort. In addition, for many applications in the feature recognition domain, the performance bottleneck occurs in the execution of matrix manipulation operations such as finding eigenvalues, performing convolutions and filtering, or calculating FFTs. Highly optimized libraries are available for these types of operations. It is highly unlikely that a genetic programming system would ever evolve such complicated operations (optimized or unoptimized) during program evolution. By starting with primitives that reflect the best practices of the domain and using the genetic programming environment as the glue, we are more likely to end up with a working program. In other words, instead of trying to evolve a horse from an amoeba, we try to breed a horse

from a good bloodline. The former process is an example of general evolution, while the latter is an example of special evolution [23].

The details of data handling and support are also critical. Recent publications using stack-based representations have attempted to expand the basic data types available in a GPL by adding separate stacks for each data type. This appears to be an extension of the direction taken by early stack based languages such as FORTH. However, a separate stack design requires adding not only a full set of data manipulation functions for each stack but also functions for data transfer between stacks. As previously indicated, this choice adversely expands the dimensionality of the program search space.

The FIFTH design simplifies the language structure by using a single data stack capable of handling multiple data types. This is accomplished by using a stack that holds containers. Containers provide strong type safety without requiring explicit type annotations or overloaded operations based on explicit data types. Rather, data types are either statically inferred or dynamically tagged. This is similar to the data model found in MATLAB. One reason that MATLAB is popular with signal processing analysts who are not programmers is that variables do not have to be statically declared, and functions automatically determine argument data types at run-time. MATLAB's rich function set together with run-time typing makes the expression of many DSP algorithms both shorter and easier to understand than the same algorithm written in C or C++. Our vision is that similar programs written in FIFTH will be even more compact.

The main disadvantage to this approach is that data handling is more complex since all data, even simple types, must be handled in a structure. Also, since this model is not closely aligned with hardware, there will be more execution overhead.

Our work so far indicates that the FIFTH architecture has the potential of making a large class of vector based feature recognition and signal processing algorithms amenable to a genetic programming approach.

References

1. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming–An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann, dpunkt.verlag, San Francisco (1998)
2. Spector, L., Klein, J., Keijzer, M.: The Push3 Execution Stack and the Evolution of Control. In: GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, Vol. 2. ACM SIGEVO (formerly ISGEC) ACM Press New York, NY, 10286-1405, USA, Washington DC, USA (2005) 1689–1696
3. Spector, L., Perry, C., Klein, J., Keijzer, M., Hampshire College School of Cognitive Science, Push 3.0 Programming Language Description. Accessed September 2005, http://hampshire.edu/lspector/push3-description.html (2003)
4. Gagn, C., Parizeau, M.: Open BEAGLE: A New C++ Evolutionary Computation Framework. In: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference. Morgan Kaufmann Publishers San Francisco, CA 94104, USA, New York (2002)
5. Perkis, T.: Stack-Based Genetic Programming. In: Proceedings of the 1994 IEEE World Congress on Computational Intelligence. IEEE Press, Orlando, Florida, USA (1994)
6. Spector, L., Robinson, A.: Genetic Programming and Autoconstructive Evolution with the Push Programming Language. In: Genetic Programming and Evolvable Machines (2002)

7. Tchernev, E.: Stack-Correct Crossover Methods in Genetic Programming. In: Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002). AAAI 445 Burgess Drive, Menlo Park, CA 94025, New York, NY (2002)
8. Silva, S.: GPLAB - A Genetic Programming Toolbox for MATLAB (2005)
9. Discipulus™ Genetic-Programming Software. RML Technologies, Inc. (2004)
10. Teller, A., Veloso, M.: Program Evolution for Data Mining. The International Journal of Expert Systems **8** (1995) 216–236
11. Sharman, K.C., Esparcia-Alcazar, A.I., Li, Y.: Evolving Digital Signal Processing Algorithms by Genetic Programming. In. Faculty of Engineering, Glasgow G12 8QQ, Scotland (1995)
12. Rizki, M.M., Tamburino, L.A.: Evolutionary Computing Applied To Pattern Recognition. In: Koza, J.R., Banzhaf, n.W., Chellapilla, n.K., Deb, n.K., Dorigo, n.M., Fogel, n.D.B., Garzon, n.M.H., Goldberg, n.D.E., Iba, n.H., Riolo, n.R. (eds.): Genetic Programming 1998: Proceedings of the Third Annual Conference. Morgan Kaufmann San Francisco, CA, USA, University of Wisconsin, Madison, Wisconsin, USA (1998) 777–785
13. Rather, E., Bradley, M.: Programming Languages - FORTH (X3J14 dpANS-6). American National Standards Institute, Inc. (1993)
14. Mammone, R.J., Rothaker, R.J., Podilchuk, C.I.: Estimation of carrier frequency, modulation type, and bit rate of an unknown modulated signal. In: IEEE International Conference on Communications, Vol. 2 (1987) 1006–1012
15. Cardelli, L.: Type Systems. In: Tucker, A.B. (ed.): CRC Handbook of Computer Science and Engineering. CRC Press, Boca Raton, FL (2004)
16. Haynes, T.D., Schoenefeld, D.A., Wainwright, R.L.: Type Inheritance in Strongly Typed Genetic Programming. In: Angeline, P.J., nd K. E. Kinnear, Jr. (eds.): Advances in Genetic Programming 2. MIT Press, Cambridge, MA, USA (1996) 359–376
17. MathWorks, The Mathworks, Inc., MAT-File Format. www.mathworks.com (2005)
18. Schmidt, D.C., Real-time CORBA with TAO. http://www.cs.wustl.edu/~schmidt/TAO.html (2006)
19. Tchernev, E.B., Phatak, D.S.: Control structures in linear and stack-based Genetic Programming. In: Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference, Seattle, Washington, USA (2004)
20. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
21. Holladay, K., Robbins, K.: A Framework for Automatic Large-Scale Testing and Characterization of Signal Processing Algorithms. In: Military Communications (MILCOM), Monterey, CA (2004)
22. Holladay, K., Robbins, K.: Experimental Analysis of Wavelet Transforms for Estimating PSK Symbol Rate. In: IASTED International Conference on Communication Systems and Applications, Banff, Canada (2004)
23. Kerkut, G.A.: Implications of Evolution. Pergamon Press Inc., New York (1960)

Genetic Programming with Fitness Based on Model Checking

Colin G. Johnson

Computing Laboratory, University of Kent, Canterbury,
Kent, CT2 7NF, England
C.G.Johnson@kent.ac.uk

Abstract. Model checking is a way of analysing programs and program-like structures to decide whether they satisfy a list of temporal logic statements describing desired behaviour. In this paper we apply this to the fitness checking stage in an evolution strategy for learning finite state machines. We give experimental results consisting of learning the control program for a vending machine.

1 Introduction

In genetic programming (and similar systems for the induction of program code), fitness is typically evaluated by running the programs on a sample of test data. This brings with it a number of difficulties[17], most notably that we cannot formally have confidence in the performance of the system for any input data that has not been used in training. In many cases this does not matter; however, in safety- and mission-critical domains, this is a significant barrier to the adoption of such techniques.

In this paper we give an example of the use of a formal reasoning technique—model checking—as a way of assessing fitness in an inductive automatic programming system. Model checking requires the user to specify the desirable qualities of a system in the form of temporal logic statements about program state; these properties can then be analysed for *all* possible program states. The end result is either a confirmation that the properties always hold, or a counterexample which demonstrates why the system under test does not satisfy the statements.

The paper is structured as follows. Section 2 reviews the literature on the application of program analysis techniques to automatic program induction. This is followed by a section which gives a brief overview of model checking, the analysis technique that we have used in the experiments in this paper. Section 4 discusses how model checking can be used as a fitness measure, and section 5 describes a novel Evolution Strategy for evolving state machines. Section 6 gives the problem specifications that are used in the experiments in the paper, and this is followed by a section which gives results. The paper finishes with some conclusions and ideas for future work.

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 114–124, 2007.
© Springer-Verlag Berlin Heidelberg 2007

2 Background - Program Induction with Guaranteed Behaviour

A number of techniques exist which generate computer programs, or other formal executable program-like structures (digital circuits, communications protocols, state machines), by an inductive process which is driven by comparisons between trial solutions and descriptions of the desired behaviour of the system. Perhaps the best known of these is *genetic programming* [2,20]. This technique applies genetic algorithms to derive executable structures by using a representation which fits well with evolutionary operators such as crossover and mutation. Other evolutionary approaches include *grammatical evolution* [21,22], where the representation is a simple string of symbols, but this is converted into a complex structure via a grammar; and *evolutionary programming* [10,11] where evolutionary operators operate directly on a state machine representation. Other heuristic search techniques, such as simulated annealing [7], have been applied to the induction of executable systems. Overall, this area has been termed *search-based software engineering* [6,13].

Typically such techniques measure the quality of the trial solution by running some instantiations of the system, measuring the results from the system, and comparing those results to the desired behaviour—essentially *testing* the system.

A small number of studies have used measures of fitness that are not dependent solely on testing-like measures of performance, but instead on some form of analysis of the program. In earlier work, these were typically used to guide the search in areas related to performance or evolvability of the solutions: for example measuring length of solutions [26] or using a metric for program complexity [12]. Clearly such measures need to be used alongside a measure of problem-solving quality.

By contrast, a small number of authors have used techniques based on the analysis of the candidate programs to measure ability to solve the problem at hand. Static analysis of programs has been applied to the induction of programs which solve sorting problems [15] and geometric placement problems [17]. Other approaches combine static analysis with data-driven analysis: Keijzer [19] uses static analysis techniques as a preprocessing step, whilst Johnson [18] has applied static analysis to impose safety constraints on programs that are otherwise data-driven.

A general motivation for this kind of approach has been suggested by Partridge and colleagues [23,24]. He notes that problems that are naturally "data-defined" (e.g. pattern recognition problems) are frequently approached by constructing an artificial "specification bottleneck" which attempts to give a specification to a problem that is best described by giving a number of examples. By contrast, in traditional GP fitness evaluation the opposite problem is sometimes observed: problems for which there is a clear specification are evaluated crudely by giving a number of test cases: this could analogously be termed a "data bottleneck".

The work described in this paper fits into an overall program of work that attempts to measure fitness in terms of the native representation, be it data or specification. Furthermore, multicriterion optimization methods can be used

to solve problems where some aspects of the problem are best described as specifications, and others as data, as illustrated in [18].

3 Background - Model Checking

Model checking [8,16] is a technique for confirming that a program satisfies a number of conditions. A model checking system takes as input two things. The first of these is a description of some aspect of the system being constructed, expressed as a statement in a temporal logic [9]; that is, statements about how the variables and states in the program change with time. The second input is a description of the system which the user believes should satisfy that description: this might be a computer program, a communications protocol, a state machine, a circuit diagram, et cetera.

The model checking program constructs an abstracted, symbolic representation of the system being analysed, and then uses this representation to decide, in an efficient manner [3], whether the statements always hold in the system, regardless of the control-flow path that is taken through the system. At the end of this analysis, the program reports either a positive result (that the system will always satisfy the statement) or a negative result together with a counterexample which gives a program path under which the statement is not satisfied.

The description language that we use in this study is CTL (Computation Tree Logic) [9]. This consists a number of basic "atomic propositions" (in the examples below, these are labels of states and values of variables in a finite state machine), which can be combined by standard propositional logic connectives and a set of *temporal* connectives which act on propositions (including propositions which themselves contain temporal connectives).

These temporal connectives consist of two components: a description of the *scope* over the future time paths (either A or E) and a description of *when* the proposition that is the argument of the temporal operator holds within that scope (one of G,F,X or U). These have (in informal terms) the following meanings:

A The proposition will hold on **All** paths starting from the current point.
E There **Exists** a path on which the proposition will hold.
G The proposition holds all states (**Globally)** along the path.
F The proposition can be found somewhere (in the **Future**) along the path.
X The **neXt** state satisfies the proposition.
U The proposition holds **Until** a second proposition holds (this is the only binary operator - the rest are unary).

Here a "path" is a sequence of states in time, starting from the current state. Some illustrative examples are given in Figure 1.

This language allows a description of how a process will change with time. Model checking algorithms automatically check whether a particular description of a system satisfies a CTL statement describing the system. The model checking algorithm used below is the SMV system (`http://www.cs.cmu.edu/~modelcheck/smv.html`), in particular the version used in the Stuttgart Model-Checking Kit (`http://www.fmi.uni-stuttgart.de/szs/tools/mckit/`).

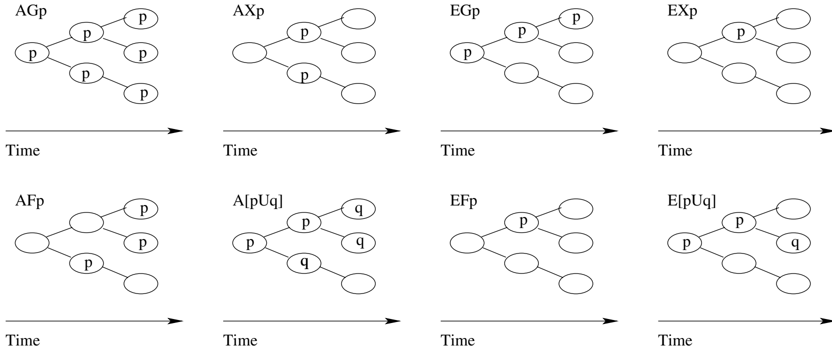


Fig. 1. Illustrative examples of the CTL operators

4 Model Checking as a Fitness Measure

In the experiments below each problem is described as a list of CTL statements that describe the desired properties of the program being evolved. The fitness is measured by the number of statements that are satisfied. To help to smooth out the fitness landscape, some statements are included in the specification lists below that act as *stages* on the way to a complete specification.

5 Methods - Induction of Communicating Finite Automata by an Accumulative Evolution Strategy

In the work in this paper we use *Communicating Finite Automata* (CFA) [4] as the description language for the structures that are evolved. We use a $(1+\lambda)$ Evolution Strategy with a novel growth-style mutation operator to evolve these structures.

A CFA consists of a number of nodes, which can be labelled from a label-set, linked by a number of directed edges. These edges can contain conditions and actions concerned with global variables and communication channels between automata: e.g. checking whether a variable is set to a particular value or within a range, whether the value of a variable is changed by executing the transition, or whether a variable reads/writes its value from/to a communication channel. Examples are given in Figure 3.

The learning process used in this paper will be referred to as an *Accumulative* Evolution Strategy (AES). This starts with very simple structures, and the most probably moves that can be made at the mutation stage consist of *adding* items to the structure. Therefore a solution to the problem is built up by accumulating substructures, rather than traditional approaches which begin with structures that are of similar complexity to the final desired structure, and where mutation is typically converting one structure into another of similar size.

The motivation for this variant on traditional evolutionary algorithms is that potential solutions that contain large numbers of arbitrarily connected nodes will fail to satisify any of the fitness conditions. This is because there will as

a consequence be a large number of routes through the (nondeterministic) automaton, and therefore there is a large chance that statements such as "X must always be followed by Y" used in the fitness checking will not be satisfied. The aim of our strategy is to build up a solution by conservatively adding new parts to the structure over evolutionary time (a similar point has been made recently by Petrovic [25].

The AES system runs as follows:

INPUT:
  List of labels for nodes
  List of variables and channels
  User model
  List of CTL statements
INITIALISE:
  Create an automaton A consisting of one node labelled "start"
LOOP: until solution found or a fixed number of timesteps completed
  Generate λ mutations $A'_{1..\lambda}$ of A by the following list of processes:
    - with a 0.4 probability, add a new (unconnected) node
      (0.5 probability of label "blank",
      otherwise labelled with a random label from the label set)
    - with a 1.0 probability (always!) add a new link
      (this link has a random label)
    - with a 0.3 probability delete a (randomly chosen) link
    - with a 0.1 probability rename a (randomly chosen) node
    - with a 0.2 probability rename a (randomly chosen) link
  Run the model checker on each statement on each of $A'_{1..\lambda}$
  Count how many statements are satisfied for each of $A'_{1..\lambda}$
  Let the new A be the member of $A'_{1..\lambda}$with the most statements satisfied
ENDLOOP
OUTPUT:
  The best solution found
  The number of generations required to find the best

In the experiments below $\lambda = 20$. The probabilities of performing the various mutations are parameters which have been empirically determined.

No recombination is used in this method at present. This remains an option for future studies—however, recombination has not been heavily used in applications of evolution to finite state automata. One significant reason for this is that there is no clear notion of what a meaningful subroutine is in order to carry our recombination—by contrast, Koza-style tree-based representations [20] have an unambiguous notion of what can be readily swapped between trees (though the actual role of recombination is controversial). Some attempts have been made [5,1] to automatically identify functionally coherent modules in automata—such modules could be usefully used as the units of recombination.

6 Methods - Some Example Specifications

In the experiments below we will generate automata that represent the control systems for coffee vending machines. We use two examples. Each problem consists

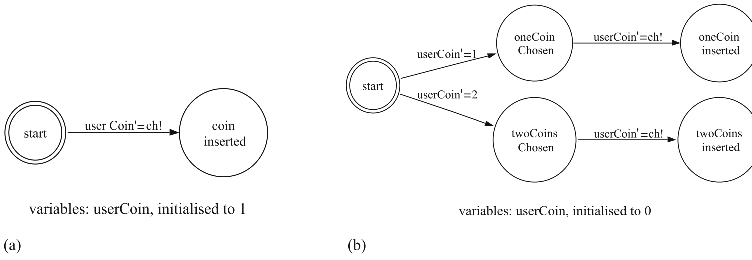


Fig. 2. Automata representing user behaviour: (a) problem 1 (b) problem 2

Table 1. The CTL specification for the coffee machine in problem 1

Variables	coin, taking values $\{0, 1\}$	
States	coffee	
	reset	
	blank	
Statements	EF("coin=1")	1. We can reach a state where the coin has been inserted
	EF("coffee")	2. We can reach a state labelled "coffee"
	EF("reset")	3. We can reach a state which are labelled "reset"
	AG("userStart"→AF("coffee"))	4. Once the user process has started, coffee must be served
	EF("coffee")&EF("reset")& AG("coffee"→EF("reset"))	5. States coffee and reset can be reached, and whenever coffee has been reached, the machine *can* reset.
	EF("coffee")&EF("reset")& AG("coffee"→AF("reset"))	6. States coffee and reset can be reached, and whenever coffee has been reached, the machine *must* reset.
	EF("coffee")&AG("coffee"→ AX(AG(!"coffee")))	7. Once coffee has been served, we must not serve another coffee
	EF("coin=0")&EF("coffee")& AG("coin=0"→AG(!"coffee"))	8. If no coin has been inserted, we cannot get a coffee

of two parts: an automaton that represents user behaviour, which is fixed for the given problem; and a specification of the desired machine behaviour, given as a sequence of CTL statements. The specification is then used to measure fitness in the learning process, which learns a machine-automaton which, accompanied by the user-automaton provided, gives a complete description of the system.

6.1 Problem 1

The first problem is a simple machine where the user places a coin into the machine, receives coffee, and the machine then goes into a reset state.

Table 2. The CTL specification for the coffee/tea machine in problem 2

Variables	coin, taking values $\{0, 1, 2\}$	
States	coffee	
	tea	
	reset	
	blank	
Statements	EF("coin=1")	1. We can reach a state where one coin has been inserted
	EF("coin=2")	2. We can reach a state where two coins have been inserted
	EF("coffee")	3. We can reach a state labelled "coffee".
	EF("tea")	4. We can reach a state labelled "tea".
	EF("reset")	5. We can reach a state labelled "reset".
	AG("userStart"→AF("coffee"|"tea"))	6. Once the user has started acting tea or coffee must be served
	AG("oneCoinSelected"→AF("coffee"))	7. Once the user has chosen to insert one coin coffee must be served
	AG("twoCoinsSelected"→AF("tea"))	8. Once the user has chosen to insert two coins tea must be served
	AG("oneCoinSelected"→AG(!"tea"))	9. Once the user has chosen to insert one coin tea must not be served
	AG("twoCoinsSelected"→AG(!"coffee"))	10. Once the user has chosen to insert two coins coffee must not be served
	EF("coffee")&EF("reset")& AG("coffee"→EF("reset"))	11. Once coffee has been served we can reset
	EF("coffee")&EF("reset")& AG("coffee"→AF("reset"))	12. Once coffee has been served we must reset
	EF("tea")&EF("reset")& AG("tea"→EF("reset"))	13. Once tea has been served we can reset
	EF("tea")&EF("reset")& AG("tea"→AF("reset"))	14. Once tea has been served we must reset
	EF("coffee")&AG("coffee"→ AX(AG(!("coffee"|"tea"))))	15. Once coffee has been served, no more coffee or tea can be served
	EF("tea")&AG("tea"→ AX(AG(!("coffee"|"tea"))))	16. Once tea has been served, no more coffee or tea can be served
	EF("coin=0")&EF("coffee")& AG("coin=0"→AG(!("coffee"|"tea")))	17. If no coin has been inserted, we cannot get a coffee or tea

The automaton representing user behaviour in this example is very simple (Figure 2a)—the user can perform one action, putting a coin in the machine, which is represented by adding a value to a *channel* (a list which can be read by other automata), which represents adding a coin into the machine.

The specification of the machine behaviour is given in table 1. Note that some of the behaviours are *staged*, e.g. rules 1–3 are concerned with making certain that certain states are accessible at all, before they are used in specific ways later on. Similarly, rule 5 is the *can* version of the *must* rule in rule 6. The aim of these is to smooth out the fitness space.

6.2 Problem 2

The second problem is a machine where the user places either one or two coins into the machine, receives coffee (for one coin) or tea (for two coins), and the machine then goes into a reset state.

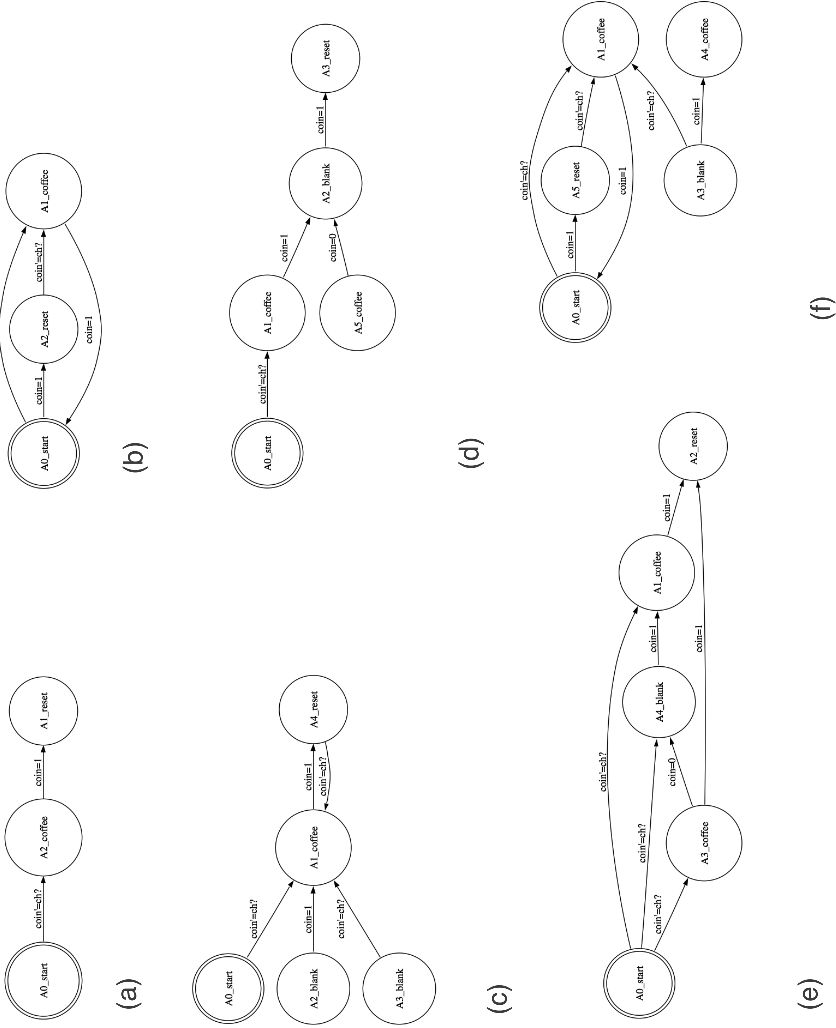


Fig. 3. Example solutions from problem 1

The automaton representing user behaviour in this example is given in Figure 2b), and the specification of the machine behaviour is given in table 2.

7 Results

In this section the results of experiments using the Accumulative Evolution Strategy on the two problems are given, and a discussion of the results made.

7.1 Experiment 1

The first experiment consisted of 30 runs of the Accumulative Evolution Strategy on problem 1 with $\lambda = 20$ with the algorithm being stopped after 20 runs if a solution satisfying all the conditions had not been found.

The algorithm found a solution that satisfied all eight conditions in 25/30 runs; in the remaining runs seven conditions were satisfied. In those cases where a solution was found, the mean number of generations needed was 6.5 (with standard deviation 4.4).

A number of examples of successful solutions can be found in Figure 3.

7.2 Experiment 2

The first experiment consisted of 30 runs of the Accumulative Evolution Strategy on problem 2 with $\lambda = 20$ with the algorithm being stopped after 30 runs if a solution satisfying all the conditions had not been found.

No run found a solution with all (seventeen) conditions satisfied. The mean number of conditions satisfied was 14 (with a standard deviation of 2). Seven runs found examples where 16 out of the 17 cases were satisfied.

8 Conclusions and Ongoing Work

We have demonstrated how model checking can be used to measure fitness in the evolution of state machines. In the future we intend to apply this to a number of other example problems, including problems which have both specification and data-driven aspects.

An important area for future work in terms of developing the technique will be to develop techniques for smoothing out the fitness landscapes. Ideas in this area include scaling (perhaps dynamically) the fitness contributions of each statement, to encourage the algorithm to search for less well represented statements; using the counterexamples that are returned from failed statements to measure how far the current attempt is from a solution; and estimating the number of paths within the model checking algorithm which do/don't satisfy the statement in order to get away from a simple yes/no response (perhaps using a probabilistic model checking system such as Prism [14]).

References

1. Ricardo Nastas Acras and Silvia Regina Vergilio. Splinter: A generic framework for evolving modular finite state machines. In *Proceedings of SBIA 2004*, pages 356–365. Springer, 2004.
2. Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming: An Introduction*. Morgan Kaufmann, 1998.
3. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990.
4. Thomas L. Casavant and Jon G. Kuhl. A communicating finite automata approach to modeling distributed computation and its application to distributed decision-making. *IEEE Trans. Computers*, 39(5):628–639, 1990.
5. Kumar Chellapilla and David Czarnecki. A preliminary investigation into evolving modular finite state machines. In *Proceedings of the 1999 IEEE Congress on Evolutionary Computation*. IEEE Press, 1999.
6. John Clark, Jose Javier Dolado, Mark Harman, Rob Hierons, Mary Lumkin Bryan Jones, Brian Mitchell, Spiros Mancoridis, Kearton Rees, Marc Roper, and Martin Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, 2003.
7. John A. Clark and Jeremy L. Jacob. Protocols are programs too: the meta-heuristic search for security protocols. *Information and Software Technology*, 43(14):891–904, 2001.
8. Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
9. E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 955–1072. MIT Press, 1990.
10. D.B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, 1995.
11. Lawrence J. Fogel, Alvin J. Owens, and Michael J. Walsh. *Artificial Intelligence through Simulated Evolution*. Academic Press, 1966.
12. Mark Harman and John A Clark. Metrics are fitness functions too. In *Proceedings of the Tenth IEEE International Symposium on Software Metrics*, pages 58–69. IEEE Press, 2004.
13. Mark Harman and Bryan F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.
14. A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: a tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proceedings 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, pages 441–444, 2006. LNCS Volume 3920.
15. Lorenz Huelsbergen. Abstract program evaluation and its application to sorter evolution. In *Proceedings of the 2000 Congress on Evolutionary Computation*, pages 1407–1414. IEEE Press, 2000.
16. Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.
17. Colin G. Johnson. Deriving genetic programming fitness properties by static analysis. In James Foster, Evelyne Lutton, Conor Ryan, and Andrea Tettamanzi, editors, *Proceedings of the 2002 European Conference on Genetic Programming*. Springer, 2002.

18. Colin G. Johnson. Genetic programming with guaranteed constraints. In Ahmad Lotfi and Jonathan M. Garibaldi, editors, *Applications and Science in Soft Computing*, pages 95–100. Springer, 2004.
19. Maarten Keijzer. Improving symbolic regression with interval arithmetic. In Conor Ryan et al., editor, *Proceedings of the 6th European Conference on Genetic Programming*, pages 70–82, 2003.
20. John R. Koza. *Genetic Programming : On the Programming of Computers by means of Natural Selection*. Series in Complex Adaptive Systems. MIT Press, 1992.
21. Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, August 2001.
22. Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer, 2003.
23. D. Partridge and A. Galton. The specification of 'specification'. *Minds and Machines*, 5(2):243–255, 1995.
24. D. Partridge and W.B. Yates. Data-defined problems and multiversion neural-net systems. *Journal of Intelligent Systems*, 7(1–2):19–32, 1997.
25. Pavel Petrovic. Comparing finite-state automata representation with gp-trees. IDI Technical report 05/2006, Norwegian University of Science and Technology, 2006.
26. Conor Ryan. Pygmies and civil servants. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, pages 243–263. MIT Press, 1994.

Geometric Particle Swarm Optimisation

Alberto Moraglio, Cecilia Di Chio, and Riccardo Poli

Department of Computer Science,
University of Essex,
Wivenhoe Park, Colchester, CO4 3SQ, UK
{amoragn,cdichi,rpoli}@essex.ac.uk

Abstract. Using a geometric framework for the interpretation of crossover of recent introduction, we show an intimate connection between particle swarm optimization (PSO) and evolutionary algorithms. This connection enables us to generalize PSO to virtually any solution representation in a natural and straightforward way. We demonstrate this for the cases of Euclidean, Manhattan and Hamming spaces.

1 Introduction

Particle swarm optimisation (PSO) [5] has traditionally been applied to continuous search spaces. Although a version of PSO for binary search spaces has been defined [4], attempts to extend PSO to richer spaces, e.g., combinatorial spaces, have not been very successful.

There are two ways of extending PSO to richer spaces. Firstly one can rethink and adapt the PSO for each new solution representation. Secondly one can define a mathematical generalisation of the notion (and motion) of particles for a general class of spaces. This second approach has the advantage that a PSO can be derived in a principled way for any search space belonging to the given class. Here we follow this approach.

In particular, we show *formally* how a general form of PSO (without the inertia term) can be obtained by using theoretical tools developed for evolutionary algorithms using geometric crossover and geometric mutation. These are representation-independent operators that generalise many pre-existing search operators for the major representations, such as binary strings [7], real vectors [7], permutations [8], syntactic trees [8] and sequences [12].

In Sec. 2, we introduce the geometric framework and introduce the notion of multi-parental geometric crossover. In Sec. 3, we recast PSO in geometric terms and generalize it to generic metric spaces. In Sec. 4, we apply these notions to the Euclidean, Manhattan and Hamming spaces. In Sec. 5, we discuss how to specialise the general PSO automatically to virtually any solution representation using geometric crossover. In Sec. 6, we present conclusions and future work.

2 Geometric Framework

Geometric operators are defined using the notions of line segment and ball. These notions and the corresponding genetic operators are well-defined once a notion

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 125–136, 2007.
© Springer-Verlag Berlin Heidelberg 2007

of distance in the search space is defined. Defining search operators as functions of the search space is opposite to the standard way where the search space is seen as a function of the search operators employed [3].

2.1 Geometric Preliminaries

The terms *distance* and *metric* denote any real valued function that conforms to the axioms of identity, symmetry and triangular inequality. A simple connected graph is naturally associated to a metric space via its *path metric*: the distance between two nodes in the graph is the length of a shortest path between the nodes. Distances arising from graphs via their path metric are called *graphic distances*. Similarly, an edge-weighted graph with strictly positive weights is naturally associated to a metric space via a *weighted path metric*.

In a metric space (S, d) a *closed ball* is a set of the form $B(x; r) = \{y \in S | d(x, y) \leq r\}$ where $x \in S$ and r is a positive real number. A *line segment* is a set of the form $[x; y] = \{z \in S | d(x, z) + d(z, y) = d(x, y)\}$ where $x, y \in S$ are called extremes of the segment. Metric ball and metric segment generalise the familiar notions of ball and segment in the Euclidean space to any metric space through distance redefinition. In general, there may be more than one shortest path (*geodesic*) connecting the extremes of a metric segment; the metric segment is the union of all geodesics.

We assign a structure to the solution set by endowing it with a notion of distance d. $M = (S, d)$ is a solution *space* and $L = (M, g)$ is the corresponding *fitness landscape*.

A family $\mathcal{X}$ of subsets of a set X is called *convexity on* X if: (C1) the empty set $\emptyset$ and the universal set X are in $\mathcal{X}$, (C2) if $\mathcal{D} \subseteq \mathcal{X}$ is non-empty, then $\bigcap \mathcal{D} \in \mathcal{X}$, and (C3) if $\mathcal{D} \subseteq \mathcal{X}$ is non-empty and totally ordered by inclusion, then $\bigcup \mathcal{D} \in \mathcal{X}$. The pair $(X, \mathcal{X})$ is called *convex structure*. The members of $\mathcal{X}$ are called *convex sets*. By the axiom (C1) a subset A of X of the convex structure is included in at least one convex set, namely X. From axiom (C2), A is included in a smallest convex set, the *convex hull* of A: $co(A) = \bigcap\{C | A \subseteq C \in \mathcal{X}\}$. The convex hull of a finite set is called a *polytope*. The axiom (C3) requires *domain finiteness* of the convex hull operator: a set C is convex iff it includes $co(F)$ for each finite subset F of C. The convex hull operator applied to set of cardinality two is called *segment operator*. Given a metric space $M = (X, d)$ the segment between a and b is the set $[a, b]_d = \{z \in X | d(x, z) + d(z, y) = d(x, y)\}$. The abstract *geodetic convexity* $\mathcal{C}$ on X induced by M is obtained as follow: a subset C of X is geodetically-convex provided $[x, y]_d \subseteq C$ for all x, y in C. If co denotes the convex hull operator of $\mathcal{C}$, then $\forall a, b \in X : [a, b]_d \subseteq co\{a, b\}$. The two operators need not to be equal: there are metric spaces in which metric segments are not all convex.

2.2 Two-Parent and Multi-parent Geometric Crossover

Definition 1. *(Geometric crossover) A binary operator is a geometric crossover under the metric d if all offspring are in the segment between its parents.*

The definition is *representation-independent* and, therefore, crossover is well-defined for any representation. Being based on the notion of metric segment, *crossover is only function of the metric d* associated with the search space.

This class of operators is really broad. For example, it includes: various types of blend or line crossovers, box recombinations, and discrete recombinations [7]; homologous crossovers [7,9]; PMX, Cycle crossover and merge crossover [8]; homologous GP crossovers [11]; and several others [12,7,8,10].

We now provide the following extension:

Definition 2. *(Multi-parental geometric crossover) In a multi-parental geometric crossover, given n parents $p_1, p_2, \ldots, p_n$ their offspring are contained in the metric convex hull of the parents $co(\{p_1, p_2, \ldots, p_n\})$ for some metric d.*

Theorem 1. *(Decomposable three-parent recombination) Every recombination $RX(p_1, p_2, p_3)$ that can be decomposed as a sequence of 2-parental geometric crossovers GX and GX' under the same metric, so that $RX(p_1, p_2, p_3) = GX(GX'(p_1, p_2), p_3)$, is a three-parental geometric crossover.*

Proof. Let P be the set of parents and $co(P)$ their metric convex hull. By definition of metric convex hull, for any two points $a, b \in co(P)$ their offspring are in the convex hull $[a, b] \subseteq co(P)$. Since $P \subseteq co(P)$, any two parents $p_1, p_2 \in P$ have offspring $o_{12} \in co(P)$. Then any other parent $p_3 \in P$ when recombined with o_{12} produces offspring o_{123} in the convex hull $co(P)$. So the three-parental recombination equivalent to the sequence of geometric crossover $GX'(p_1, p_2) \rightarrow o_{12}$ and $GX(o_{12}, p_3) \rightarrow o_{123}$ is a multi-parental geometric crossover.

3 Geometric PSO

3.1 Basic, Canonical PSO Algorithm and Geometric Crossover

Consider the canonical PSO in Algorithm 1. It is well known that one can write the equation of motion of the particle without making explicit use of its velocity.

Let x be the position of a particle and v be its velocity. Let $\hat{x}$ be the current best position of the particle and let $\hat{g}$ be the global best. Let v' and v'' be the velocity of the particle and $x' = x + v$ and $x'' = x' + v'$ its position at the next two time ticks. The equation of velocity update is the linear combination: $v' = w_1 v + w_2(\hat{x} - x') + w_3(\hat{g} - x')$ where w_1, w_2 and w_3 are scalar coefficients. To eliminate velocities we substitute the identities $v = x' - x$ and $v' = x'' - x'$ in the equation of velocity update and rearrange it to obtain an equation that expresses x'' as function of x and x': $x'' = (1 + w_1 - w_2 - w_3)x' - w_1 x + w_2 \hat{x} + w_3 \hat{g}$.

If we set $w_1 = 0$, which corresponds to setting $\omega = 0$ (i.e., the particle has no inertia), x'' becomes independent on its position two time ticks earlier x. The equation of motion becomes:

$$x'' = (1 - w_2 - w_3)x' + w_2 \hat{x} + w_3 \hat{g}. \quad (1)$$

In these conditions, the main feature that allows the motion of particles is the ability to perform linear combinations of points in the search space. As we will see

in the next section, we can achieve this same ability by using multiple (geometric) crossover operations. This makes it possible to obtain a generalisation of PSO to generic search spaces.

Algorithm 1. Standard PSO algorithm

1: **for all** particle i **do**
2: initialise position x_i and velocity v_i
3: **end for**
4: **while** stop criteria not met **do**
5: **for all** particle i **do**
6: set personal best $\hat{x}_i$ as best position found so far by the particle
7: set global best $\hat{g}$ as best position found so far by the whole swarm
8: **end for**
9: **for all** particle i **do**
10: update velocity using equation

$$v_i(t+1) = \omega v_i(t) + \phi_1 U(0,1)(\hat{g}(t) - x_i(t)) + \phi_2 U(0,1)(\hat{x}_i(t) - x_i(t)) \quad (2)$$

11: update position using equation

$$x_i(t+1) = x_i(t) + v_i(t+1) \quad (3)$$

12: **end for**
13: **end while**

3.2 Geometric Interpretation of Linear Combinations

If $v_1, ..., v_n$ are vectors and $a_1, ..., a_n$ are scalars, then the *linear combination* of those vectors with those scalars as coefficients is : $a_1v_1 + a_2v_2 + a_3v_3 + \cdots + a_nv_n$. A linear combination on n linearly independent vectors spans completely an n-dimensional space but not a higher dimensional one. So, the linear combination of three linearly independent points spans a 3-dimensional space but not a 4-dimensional one.

An *affine combination* of vectors $v_1, ..., v_n$ is a linear combination $\sum_i a_i \cdot x_i$ in which $\sum_i a_i = 1$. When a vector represents a point in space, the affine combination of 2 independent points spans completely the line passing through them; the affine combination of 3 points spans completely the plane (2D line) passing through them; increasing number of linearly independent points span completely higher dimensional "lines".

A *convex combination* is an affine combination of vectors where all coefficients are non-negative. It is called "convex combination", since, when vectors represent points in space, the set of all convex combinations constitutes the convex hull.

A special case is $n = 2$, where a point formed by the convex combination will lie on a straight line between two points. For three points, their convex hull is the triangle with the points as vertices.

Theorem 2. *In a PSO with no inertia ($\omega = 0$) and where learning rates are such that $\phi_1 + \phi_2 \leq 1$, the next position of a particle x' is within convex hull formed by its current position x, its local best $\hat{x}$ and the swarm best $\hat{g}$.*

Proof. As we have seen in Sec. 3.1, when $\omega = 0$, a particle's update equation becomes the linear combination in equation (1). Notice that this is an affine combination since the coefficients of x', $\hat{x}$ and $\hat{g}$ add up to 1. *Interestingly, this means that the new position of the particle is coplanar with x', $\hat{x}$ and $\hat{g}$*. If we restrict w_2 and w_3 to be positive and their sum to be less than 1, equation (1) becomes a convex combination. *Geometrically this means that the new position of the particle is in the convex hull formed by (or more informally, between) its previous position, its local best and the swarm best.*

In the next section, we generalize this simplified form of PSO from real vectors to generic metric spaces. Mutation will be required to extend the search beyond the convex hull.

3.3 Convex Combinations in Metric Spaces

Linear combinations are well-defined for vector spaces, algebraic structures endowed with scalar product and vectorial sum. A metric space is a set endowed with a notion of distance. The set underlying a metric space does not normally come with well-defined notions of scalar product and sum among its elements. So a linear combination of its elements is not defined. How can we then define a convex combination in a metric space? Vectors in a vector space can easily be understood as points in a metric space. However, the interpretation of scalars is not as straightforward: what do the scalar weights in a convex combination mean in a metric space?

As seen in Sec. 3.2, a convex combination is an algebraic description of a convex hull. However, even if the notion of convex combination is not defined for metric spaces, convexity in metric spaces is still well-defined through the notion of metric convex set that is a straightforward generalization of traditional convex set. Since convexity is well-defined for metric spaces, we still have hope to generalize the scalar weights of a convex combination trying to make sense of them in terms of distance.

The weight of a point in a convex combination can be seen as a measure of relative linear attraction toward its corresponding point versus attractions toward the other points of the combination. The closer the weight to one, the stronger the attraction to its corresponding point. The point resulting from a convex combination can be seen as the equilibrium point of all the attraction forces. The distance between the equilibrium point and a point of the convex combination is therefore a decreasing function of the level of attraction (weight) of the point: the stronger the attraction, the smaller its distance to the equilibrium point. This observation can be used to reinterpret the weights of a convex combination in a metric space as follows: $y = w_1x_1 + w_2x_2 + w_3x_3$ with w_1, w_2 and w_3 greater than zero and $w_1 + w_2 + w_3 = 1$ is generalized to mean that y is a point such that $d(x_1, y) \sim 1/w_1$, $d(x_2, y) \sim 1/w_2$ and $d(x_3, y) \sim 1/w_3$.

This definition is formal and valid for all metric spaces but it is non-constructive. In contrast a convex combination, not only defines a convex hull, but it tells also how to reach all its points. So, how can we actually pick a point in the convex hull respecting the above distance requirements? Geometric crossover will help us with this, as we show in the next section.

The requirements for a convex combination in a metric space are:

1. Convex weights: the weights respect the form of a convex combination: $w_1, w_2, w_3 > 0$ and $w_1 + w_2 + w_3 = 1$
2. Convexity: the convex combination operator combines x_1, x_2 and x_3 and returns a point in their metric convex hull, or simply triangle, under the metric of the space considered
3. Coherence between weights and distances: the distances to the equilibrium point are decreasing functions of their weights
4. Symmetry: the same value assigned to w_1, w_2 or w_3 has the same effect (so in a equilateral triangle, if the coefficients have all the same value, the distances to the equilibrium point are the same).

3.4 Geometric PSO Algorithm

The generic Geometric PSO algorithm is illustrated in Algorithm 2. This differs from the standard PSO (Algorithm 1) in that: there is no velocity, the equation of position update is the convex combination, there is mutation and the parameters w_1, w_2, and w_3 are non-negative and add up to one. The specific PSO for the Euclidean, Manhattan and Hamming spaces use the randomized convex combination operators described in Sec. 4 and space-specific mutations.

Algorithm 2. Geometric PSO algorithm

```
1: for all particle i do
2:    initialise position x_i at random in the search space
3: end for
4: while stop criteria not met do
5:    for all particle i do
6:       set personal best x̂_i as best position found so far by the particle
7:       set global best ĝ as best position found so far by the whole swarm
8:    end for
9:    for all particle i do
10:      update position using a randomized convex combination
```

$$x_i = CX((x_i, w_1), (\hat{g}, w_2), (\hat{x}_i, w_3)) \quad (4)$$

```
11:      mutate x_i
12:   end for
13: end while
```

4 Geometric PSO for Specific Spaces

4.1 Euclidean Space

Geometric PSO for the Euclidean space is not an extension of the traditional PSO. We include it to show how the general notions introduced in the previous section materialize in a familiar context. The convex combination operator for the Euclidean space is the traditional convex combination that produces points in the traditional convex hull.

In Sec. 3.3, we have mentioned how to interpret the weights in a convex combination in terms of distances. In the following we show analytically how the weights of a convex combination affect the relative distances to the equilibrium point. In particular we show that the relative distances are decreasing functions of the corresponding weights.

Theorem 3. *In a convex combination, the distances to the equilibrium point are decreasing functions of the corresponding weights.*

Proof. Let a, b and c be three points in $\mathbb{R}^n$ and $x = w_a a + w_b b + w_c c$ be a convex combination. Let us now decrease w_a to $w'_a = w_a - \Delta$ such that w'_a, w'_b and w'_c still form a convex combination and that the relative proportions of w_b and w_c remain unchanged: $\frac{w'_b}{w'_c} = \frac{w_b}{w_c}$. This requires w'_b and w'_c to be $w'_b = w_b(1+\Delta/(w_b+w_c))$ and $w'_c = w_c(1+\Delta/(w_b+w_c))$. The equilibrium point for the new convex combination is $x' = (w_a - \Delta)a + w_b(1 + \Delta/(w_b + w_c))b + w_c(1 + \Delta/(w_b + w_c))c$. The distance between a and x is $|a - x| = |w_b(a - b) + w_c(a - c)|$ and the distance between a and the new equilibrium point is $|a - x'| = |w_b(1 + \Delta/(w_b + w_c))(a - b) + w_c(1 + \Delta/(w_b + w_c))(a - c)| = (1 + \Delta/(w_b + w_c))|a - x|$. So when w_a decreases ($\Delta > 0$) and w_b and w_c maintain the same relative proportions, the distance between the point a and the equilibrium point x increases ($|a - x'| > |a - x|$). Hence the distance between a and the equilibrium point is a decreasing function of w_a. For symmetry this applies to the distances between b and c and the equilibrium point: they are decreasing functions of their corresponding weights w_b and w_c, respectively.

The traditional convex combination in the Euclidean space respects the four requirements for a convex combination presented in Sec. 3.3.

4.2 Manhattan Space

In the following we first define a multi-parental recombination for the Manhattan space and then prove that it respects the four requirements for being a convex combination presented in Sec. 3.3.

Definition 3. *(Box recombination family) Given two parents a and b in $\mathbb{R}^n$, a box recombination operator returns offspring o such that $o_i \in [min(a_i, b_i), max(a_i, b_i)]$ for $i = 1 \dots n$.*

Theorem 4. *(Geometricity of box recombination) Any box recombination is geometric crossover under Manhattan distance*

Theorem 4 is an immediate consequence of the product geometric crossover theorem.

Definition 4. *(Three-parent Box recombination family) Given three parents a, b and c in* $\mathbb{R}^n$*, a box recombination operator returns offspring o such that* $o_i \in [min(a_i, b_i, c_i), max(a_i, b_i, c_i)]$ *for* $i = 1 \ldots n$*.*

Theorem 5. *(Geometricity of three-parent box recombination) Any three-parent box recombination is geometric crossover under Manhattan distance.*

Proof. We prove it by showing that any multi-parent box recombination $BX(a, b, c)$ can be decomposed as a sequence of two simple box recombinations. Since simple box recombination is geometric (Theorem 4), this theorem is a simple corollary of the multi-parental geometric decomposition theorem (Theorem 1).

We will show that $o' = BX(a, b)$ followed by $BX(o', c)$ can reach any offspring $o = BX(a, b, c)$. For each i we have $o_i \in [min(a_i, b_i, c_i), max(a_i, b_i, c_i)]$. Notice that $[min(a_i, b_i), max(a_i, b_i)] \cup [min(a_i, c_i), max(a_i, c_i)] = [min(a_i, b_i, c_i), max(a_i, b_i, c_i)]$. We have two cases: (i) $o_i \in [min(a_i, b_i), max(a_i, b_i)]$ in which case o_i is reachable by the sequence $BX(a, b)_i \to o_i, BX(o, c)_i \to o_i$; (ii) $o_i \notin [min(a_i, b_i), max(a_i, b_i)]$ then it must be in $[min(a_i, c_i), max(a_i, c_i)]$ in which case o_i is reachable by the sequence $BX(a, b)_i \to a_i, BX(a, c)_i \to o_i$

Definition 5. *(Weighted multi-parent Box recombination) Given three parents a, b and c in* $\mathbb{R}^n$ *and weights* w_a*,* w_b *and* w_c*, a weighted box recombination operator returns offspring o such that* $o_i = w_{a_i} a_i + w_{b_i} b_i + w_{c_i} c_i$ *for* $i = 1 \ldots n$*, where* w_{a_i}*,* w_{b_i} *and* w_{c_i} *are a convex combination of randomly perturbed weights with expected values* w_a*,* w_b *and* w_c*.*

The difference between box recombination and linear recombination (Euclidean space) is that in the latter the weights w_a, w_b and w_c are randomly perturbed only once and the same weights are used for all the dimensions, whereas the former one has a different randomly perturbed version of the weights for each dimension.

The weighted multi-parent box recombination belongs to the family of multi-parent box recombination because $o_i = w_{a_i} a_i + w_{b_i} b_i + w_{c_i} c_i \in [min(a_i, b_i, c_i), max(a_i, b_i, c_i)]$ for $i = 1 \ldots n$, hence it is geometric.

Theorem 6. *(Coherence between weights and distances) In weighted multi-parent box recombination, the distances of the parents to the expected offspring are decreasing functions of the corresponding weights.*

The proof of theorem 6 is a simple variation of that of theorem 3.

In summary in this section we have introduced the weighted multi-parent box recombination and shown that it is a convex combination operator satisfying the four requirements of a metric convex combination for the Manhattan space: convex weights by definition (Definition 4), convexity (geometricity, Theorem 5), coherence (Theorem 6) and symmetry (self-evident).

4.3 Hamming Space

In the following we first define a multi-parental recombination for binary strings that is a straightforward generalization of mask-based crossover with two parents and then prove that it respects the four requirements for being a convex combination in the Hamming space presented in Sec. 3.3.

Definition 6. *(Three-parent mask-based crossover family) Given three parents* a, b *and* c *in* $\{0,1\}^n$, *generate randomly a crossover mask of length* n *with symbols from the alphabet* $\{\mathtt{a},\mathtt{b},\mathtt{c}\}$. *Build the offspring* o *filling each position with the bit from the parent appearing in the crossover mask at the position.*

The weights w_a, w_b and w_c of the convex combination indicate for each position in the crossover mask the probability of having the symbols `a`, `b` or `c`.

Theorem 7. *(Geometricity of three-parent mask-based crossover) Any three-parent mask-based crossover is geometric crossover under Hamming distance.*

Proof. We prove it by showing that any three-parent mask-based crossover can be decomposed as a sequence of two simple mask-based crossovers. Since simple mask-based crossover is geometric, this theorem is a simple corollary of the multi-parental geometric decomposition theorem (Theorem 1).

Let m_{abc} the mask to recombine a, b and c producing the offspring o. Let m_{ab} the mask obtained by substituting all occurrences of `c` in m_{abc} with `b` and m_{bc} the mask obtained by substituting all occurrences of `a` in m_{abc} with `b`. Now recombine a and b using m_{ab} obtaining b'. Then recombine b' and c using m_{bc} where the `b`'s in the mask stand for alleles in b'. The offspring produced by the second crossover is o, so the sequence of the two simple crossovers is equivalent to the three-parent crossover. This is because the first crossover passes to the offspring the all genes it needs to take from a according to m_{abc} and the rest of the genes are all from b; the second crossover corrects those genes that should have been taken from parent c according to m_{abc} but were taken from b instead.

Theorem 8. *(Coherence between weights and distances) In weighted three-parent mask-based crossover, the distances of the parents to the expected offspring are decreasing functions of the corresponding weights.*

Proof. We want to know the expected distance from parent p_1, p_2 and p_3 and their expected offspring o as a function of the weights w_1, w_2 and w_3. To do so, we first determine, for each position in the offspring, the probability to be the same as p_1. From that then we can easily compute the expected distance between p_1 and o. We have that

$$pr\{o = p_1\} = pr\{p_1 \to o\} + pr\{p_2 \to o\} \cdot pr\{p_1|p_2\} + pr\{p_3 \to o\} \cdot pr\{p_1|p_3\} \quad (5)$$

where: $pr\{o = p_1\}$ is the probability of a bit of o at a certain position to be the same as the bit of p_1 at the same position; $pr\{p_1 \to o\}$, $pr\{p_2 \to o\}$ and $pr\{p_3 \to o\}$ are the probabilities that a bit in o is taken from parent p_1, p_2 and p_3,

respectively (these coincide with the weights of the convex combination w_1, w_2 and w_3); $pr\{p_1|p_2\}$ and $pr\{p_1|p_3\}$ are the probabilities that a bit taken from p_2 or p_3 coincides with the one in p_1 at the same location. These last two probabilities equal the number of common bits in p_1 and p_2 (p_1 and p_3) over the length of the strings n. So $pr\{p_1|p_2\} = 1 - H(p_1, p_2)/n$ and $pr\{p_1|p_3\} = 1 - H(p_1, p_3)/n$ where $H(\cdot, \cdot)$ is the Hamming distance. So equation (5) becomes

$$pr\{o = p_1\} = w_1 + w_2(1 - H(p_1, p_2)/n) + w_3(1 - H(p_1, p_3)/n). \tag{6}$$

Hence the expected distance between the parent p_1 and the offspring o is: $E(H(p_1, o)) = n \cdot (1 - pr\{o = p_1\}) = w_2 H(p_1, p_2) + w_3 H(p_1, p_3)$. Notice that this is a decreasing function of w_1 because increasing w_1 forces w_2 or w_3 to decrease since the sum of the weights is constant, hence $E(H(p_1, o))$ decreases. Analogously, $E(H(p_2, o))$ and $E(H(p_3, o))$ are decreasing functions of their weights w_2 and w_3, respectively.

In summary in this section we have introduced the weighted multi-parent mask-based crossover and shown that it is a convex combination operator satisfying the four requirements of a metric convex combination for the Hamming space: convex weights by definition (Definition 5) , convexity (geometricity, Theorem 7), coherence (Theorem 8) and symmetry (self-evident).

5 Towards a Geometric PSO for GP and Other Representations

Before looking into how we can extend geometric PSO to other solution representations, we will discuss the relation between 3-parental geometric crossover and the symmetry requirement for a convex combination.

For each of the spaces considered in section 4, we have first considered, or defined, a three-parental recombination and then we proved that it is a three-parental geometric crossover by showing that it can actually be decomposed into two sequential applications of a geometric crossover for the specific space.

However, we could have skipped altogether the *explicit definition* of a three-parental recombination. In fact to obtain the three-parental recombination we could have used two sequential applications of a known two-parental geometric crossover for the specific space. This composition is indeed a three-parental recombination, it combines three parents, and it is decomposable by construction, hence it is a three-parental geometric crossover. This, indeed, would have been simpler than the route we took.

The reason we preferred to define explicitly a three-parental recombination is that the requirement of symmetry of the convex combination is true by construction: if the roles of any two parents are swapped exchanging in the three-parental recombination both positions and respective recombination weights, the resulting recombination operator is equivalent to the original operator.

The symmetry requirement becomes harder to enforce and prove for a three-parental geometric crossover obtained by two sequential applications of a two-parental geometric crossover. We illustrate this in the following. Let us consider

three parents a, b and c with positive weights w_a, w_b and w_c which add up to one. If we have a symmetric three-parental weighted geometric crossover ΔGX, the symmetry of the recombination is guaranteed by the symmetry of the operator. So, $\Delta GX((a,w_a),(b,w_b),(c,w_c))$ is equivalent to $\Delta GX((b,w_b),(a,w_a),(c,w_c))$, hence the requirement of symmetry on the weights of the convex combination holds. If we consider a three-parental recombination defined by using twice a two-parental genetic crossover GX we have:

$$\Delta GX((a,w_a),(b,w_b),(c,w_c)) = GX((GX((a,w'_a),(b,w'_b)),w_{ab}),(c,w'_c)) \quad (7)$$

with the constraint that w'_a and w'_b are positive and add up to one and w_{ab} and w'_c are positive and add up to one. It is immediate to notice the inherent asymmetry in this expression: the weights w'_a and w'_b are not directly comparable with w'_c because they are relative weights between a and b. Moreover there is the extra weight w_{ab}. This makes the requirement of symmetry problematic to meet: given the desired w_a, w_b and w_c, what values of w'_a, w'_b, w_{ab} and w'_c do we have to choose to obtain an equivalent symmetric 3-parental weighted recombination expressed as a sequence of two two-parental geometric crossovers?

For the Euclidean space, it is easy to answer this question using simple algebra: $\Delta GX = w_a \cdot a + w_b \cdot b + w_c \cdot c = (w_a + w_b)(\frac{w_a}{w_a+w_b} \cdot a + \frac{w_b}{w_a+w_b} \cdot b) + w_c \cdot c$. Since the convex combination of two points in the Euclidean space is $GX((x,w_x),(y,w_y)) = w_x \cdot x + w_y \cdot y$ and $w_x, w_y > 0$ and $w_x + w_y = 1$ then $\Delta GX((a,w_a),(b,w_b),(c,w_c)) = GX((GX((a,\frac{w_a}{w_a+w_b}),(b,\frac{w_b}{w_a+w_b})),w_a + w_b),(c,w_c))$. This question may be less straightforward to answer for other spaces, although we could use the equation above as a rule-of-thumb to map the weights of ΔGX and the weights in the sequential GX decomposition.

Where does this discussion leave us in relation to the extension of geometric PSO to other representations? We have seen that there are two alternative ways to produce a convex combination for a new representation: (i) explicitly define a symmetric three-parental recombination for the new representation and then prove its geometricity by showing that it is decomposable into a sequence of two two-parental geometric crossovers, or (ii) use twice the simple geometric crossover to produce a symmetric or nearly symmetric three-parental recombination. In this paper we used the first approach, but the second option is also very interesting because *it allows us to extended automatically geometric PSO to all representations we have geometric crossovers for, such as permutations, GP trees, variable-length sequences, to mention a few, and virtually any other complex solution representation.*

6 Conclusions and Future Work

We have extended the geometric framework with the notion of multi-parent geometric crossover that is a natural generalization of two-parental geometric crossover: offspring are in the convex hull of the parents. Then, using the geometric framework, we have shown an intimate relation between a simplified form of PSO, without the inertia term, and evolutionary algorithms. This has enabled

us to generalize in a natural, rigorous and automatic way PSO for any type of search space for which a geometric crossover is known.

We have specialised the general PSO to Euclidean, Manhattan and Hamming spaces, obtaining three instances of the general PSO for the specific spaces.

In future work we will consider geometric PSO for permutation spaces and spaces of genetic programs, for which several geometric crossovers exist. We will also test the geometric PSO experimentally.

References

1. T. Bäck and D. B. Fogel and T. Michalewicz. Evolutionary Computation 1: Basic Algorithms and Operators. *Institute of Physics Publishing, 2000.*
2. M. Deza and M. Laurent. Geometry of cuts and metrics. *Springer, 1991.*
3. T. Jones. *Evolutionary Algorithms, Fitness Landscapes and Search.* PhD thesis, University of New Mexico, 1995.
4. J. Kennedy and R.C. Eberhart, *A Discrete Binary Version of the Particle Swarm Algorithm*, IEEE, 1997.
5. J. Kennedy and R.C. Eberhart, *Swarm Intelligence*, Morgan Kaufmann, 2001.
6. S. Luke and L. Spector. A Revised Comparison of Crossover and Mutation in Genetic Programming. *Proceedings of Genetic Programming Conference, 1998.*
7. A. Moraglio and R. Poli. Topological interpretation of crossover. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1377–1388, 2004.
8. A. Moraglio and R. Poli. Topological crossover for the permutation representation. In *GECCO 2005 Workshop on Theory of Representations*, 2005.
9. A. Moraglio and R. Poli. Product Geometric Crossover. In *Proceedings of the Parallel Problem Solving from Nature Conference*, pages 1018–1027, 2006.
10. A. Moraglio and R. Poli. Geometric Crossover for Sets, Multisets and Partitions. In *Proceedings of the Parallel Problem Solving from Nature Conference*, pages 1038–1047, 2006.
11. A. Moraglio and R. Poli. Geometric Landscape of Homologous Crossover for Syntactic Trees. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, pages 427–434, 2005.
12. A. Moraglio, R. Poli, and R. Seehuus. Geometric crossover for biological sequences. In *Proc. of European Conference on Genetic Programming*, pages 121–132, 2006.
13. P. M. Pardalos and M.G.C. Resende. Handbook of Applied Optimization. *Oxford University Press, 2002.*
14. N. Radcliffe. Equivalence Class Analysis of Genetic Algorithms. *Journal of Complex Systems, 1991*, vol. 5, pages 183–205.
15. M. L. J. Van de Vel. Theory of Convex Structures. *North-Holland, 1993* .

GP Classifier Problem Decomposition Using First-Price and Second-Price Auctions

Peter Lichodzijewski and Malcolm I. Heywood

Faculty of Computer Science, Dalhousie University
6050 University Avenue
Halifax, NS, Canada, B3H 1W5
{piotr,mheywood}@cs.dal.ca

Abstract. This work details an auction-based model for problem decomposition in Genetic Programming classification. The approach builds on the population-based methodology of Genetic Programming to evolve individuals that bid high for patterns that they can correctly classify. The model returns a set of individuals that decompose the problem by way of this bidding process and is directly applicable to multi-class domains. An investigation of two auction types emphasizes the effect of auction design on the properties of the resulting solution. The work demonstrates that auctions are an effective mechanism for problem decomposition in classification problems and that Genetic Programming is an effective means of evolving the underlying bidding behaviour.

1 Introduction

Genetic Programming (GP) is a population-based search algorithm that classically produces a single 'super' individual by way of a solution [5]. This is a natural effect of the survival of the fittest mechanism implicit in GP and is supported by various theoretical models [7]. The success of a single individual, however, may be limited in scenarios where progress cannot be made without effective problem decomposition. Attempts have been made to encourage GP to provide multiple solutions where these have typically taken the form of diversity maintenance schemes such as niching [10] and coevolution [11]. In this work we take a different approach motivated by the use of market mechanisms in machine learning, and in particular, the Hayek framework of Baum [1] [2]. However, it is apparent that the sheer number of problem-specific parameters endemic to the Hayek model results in a system that is very difficult to replicate [6].

The motivation of the current work is therefore to revisit the market-based approach for problem decomposition with the objective of keeping the model as simple as possible. We begin by considering the problem domain to be discrete, in this case, binary and multi-class classification problems. This implies that actions are also discrete so that individuals may concentrate on identifying an appropriate bidding strategy. Such a strategy need only be profitable, thus individuals are free to identify the subset of training exemplars on which they will concentrate their resources. Key to encouraging the identification of bidding

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 137–147, 2007.
© Springer-Verlag Berlin Heidelberg 2007

strategies that facilitate problem decomposition is the definition of an appropriate auction mechanism. To this end we concentrate on the design of the auction model where the auction is central to establishing the credit assignment process; if the auction is effective in instigating the relevant reward mechanism then the learning problem as a whole should also be more straightforward.

Auctions have successfully been applied to other problem domains such as the coordination of teams of robots [4]. This work demonstrates that a market-based framework using auctions can also be applied to the machine learning problem of classification. In particular, it is shown that GP is an effective means of producing the underlying bidding behaviour with minimal *a priori* knowledge.

This paper is organized as follows. The following section discusses related systems and motivates the need for a simplification to the Hayek model. Section 3 describes the proposed approach including the two types of auctions that are investigated. Section 4 summarizes model performance on four real-world datasets (two binary and two multi-class), and concluding remarks are made in Section 5.

2 Related Work

The proposed approach is motivated by the market-based Hayek model [1][2] originally applied to a blocks world problem (i.e., reinforcement learning). Hayek was found to be exceptional because it discovered generic solutions capable of producing long chains of actions. In contrast, GP augmented with hand-crafted features was able to solve problems limited in size to at most five blocks [5].

Hayek employs auctions in which individuals bid for the right to act on the environment. Because individuals must pay out their bid, good behaviours will tend to extract sufficient reward and therefore earn wealth while poor behaviours will tend to lose wealth; wealth is therefore treated as a fitness measure and used to guide the search. The key differences between Hayek and the proposed approach are as follows:

1. In Hayek, an individual always bids the same amount (limited only by the individual's wealth) yet its choice of action is a function of the input. Individuals in the proposed approach may bid different amounts, are always associated with a single action, and may posses negative wealth.
2. The Hayek population is of a variable size. Individuals are created when existing individuals accumulate enough wealth and removed when their wealth falls below a threshold. In the proposed approach, the population size is fixed and the poorest individual is always removed so that survival of the fittest is reinforced and the overhead of maintaining large populations is reduced[1].
3. In Hayek, when a child is created it receives a fraction of its parent's wealth. In return, the offspring must periodically pay to its parent a constant amount plus a fraction of its profit. In the proposed approach, there are no such transfers of wealth thus avoiding the associated problems of parameterizing the payoff feedback loop.

[1] Populations with tens of thousands of individuals were encountered in Hayek.

4. The Hayek framework taxes each individual in proportion to its size and the total computational overhead incurred by the system. The proposed approach does not use taxation. However, a fixed-size population with a steady-state selection policy ensures that the worst performing individuals do not linger in the population.

The Hayek framework was found to be complex and sensitive to a large number of paremeter setings making it difficult to reproduce previous results [6]. The design of the proposed approach results in a simpler model enabling us to concentrate on establishing the contribution of specific system components (e.g., auctions). In addition, as a starting step and to make the analysis more manageable, here the approach is applied to classification problems only.

The approach presented in this work is also related to Learning Classifier Systems [8]. Whereas classifier systems evolve populations of condition-action-strength rules, in the proposed approach the condition and strength component has been replaced by the bid procedure. This has several implications, for example, it makes the behaviours of each individual deterministic as it does not depend on any dynamic parameters such as strength. Furthermore, in the proposed approach the action set always consists of a single individual making allocation of credit more straightforward. Finally, compared to some popular classifier system formulations, in the proposed approach an individual's fitness is not solely a function of its bid accuracy [14]. Instead, individuals may survive in the population so long as their ratio of the gains made on profitable auctions to the losses sustained during unprofitable auctions is sufficiently high.

3 Methodology

A population of individuals each defining a *bid* and an *action* is evolved. The view is taken that only the bidding behaviour needs be represented as a program. The corresponding action is defined by a scalar selected *a priori* over the range of class labels, i.e., the set of integers $\{0, ..., n-1\}$ in an n-class classification problem. Given that a market model will be utilized as the methodology for problem decomposition, wealth should reflect the success of an individual's bidding behaviour. Thus, when new individuals are initialized in the population, they assume the same wealth as the poorest individual in the population. Such a view is taken in order to avoid injecting disproportionally high volumes of wealth into the market. Moreover, the initial population possesses zero wealth. This does not preclude individuals from bidding as negative wealth is possible. Thus, wealth is used as a relative measure of performance as opposed to having any monetary properties.

The following sections describe the generic auction model, two specific auction types, and the form of GP employed.

3.1 Generic Auction Model

The population is of a fixed size. During initialization, Step 1 of Algorithm 1, individuals generated with uniform probability are added to the population until

Algorithm 1. Generic auction-based evolutionary training algorithm.

1. while less than the population size limit do
   (a) seed = newRandomIndividual();
   (b) seed.wealth = 0;
   (c) population.insert(seed);
2. for each epoch do
   (a) for each training exemplar 'p' do
      i. auction(population, p);
   (b) population.delete(findPoorestIndividual(population));
   (c) parent = selectRandomIndividual(population);
   (d) child = newChild(parent);
   (e) child.wealth = findPoorestIndividual(population).wealth;
   (f) population.insert(child);

the population limit is reached. Initial wealth values are set to zero. Following initialization, the training algorithm proceeds in a series of epochs, Step 2. In the first stage of an epoch, an auction is held for each pattern in the training dataset, Step 2(a)*i*. During the auction, agents compete for the ownership of the pattern and wealths are adjusted to reflect the outcome of this competition. Reproduction takes place once the auctions have completed. At this time, the individual with the least amount of wealth is replaced. A single parent is selected with uniform probability from the population, Step 2(c), and a child is created from this parent through the application of mutation operators. The wealth of the poorest agent in the remaining population is determined, with the wealth of the child taking this value, Step 2(e). The child is then added to the population. If the new individual is profitable, its ranking with respect to wealth should 'bubble up' relative to the performance of the population in successive epochs.

The goal of training is to produce a population where each individual wins a subset of the training exemplars for which its action is suitable (e.g., in classification, suitable means that the label of the exemplar matches the action of the individual). Following training, the aggregate bidding behaviour of the population will determine how the system acts on an unseen exemplar.

3.2 Auction Types

Two types of auctions, Step 2(a)*i* of Algorithm 1, were investigated. The first represents a vanilla first-price auction in which the winning agent pays the difference between its bid and the reward resulting from its action. The second auction model explicitly encourages individuals representing different actions to minimize their bid values when their actions do not match that of the class label.

First-Price (FP). An exemplar is presented and each individual submits an associated bid. The individual with the highest bid is selected as the auction winner and must pay its bid to the environment (i.e., a payment is made

but never collected). The winner's action is then compared to the exemplar label. When the the two match, the winner receives a reward of 1, otherwise, the winner receives a reward of 0.

Second-Price (SP). Individuals again submit a bid for each exemplar and the highest bidder is selected as the auction winner. The winner's action is compared to that of the exemplar, and the FP scheme followed if the winner's action does *not* match that of the label. If the winner's action matches the label, then the highest bidder is identified from individuals representing alternative actions. The winner does not pay out its bid (to the environment) but rather the bid of this runner-up. In addition, the runner-up pays its bid (to the environment). Since the winner's action matched the class of the pattern, the winner receives a reward of 1.

In both auctions, an individual can profit only by winning auctions for patterns whose class is the same as its action without overbidding. As such, the goal is to evolve *winning* bids that reflect the reward associated with individuals' actions on each pattern. In addition, the SP auction is designed to increase the winner's profit (especially during the later stages of training when high bids are expected) and produce more robust behaviours by driving down bids of non-winning classes.

3.3 Linear Genetic Programming

Bid procedures were evolved using a linear GP representation [3]. The Sigmoid function $f(y) = (1 - e^{-y})^{-1}$ was used to obtain a bid over the unit interval given a raw (real-valued) GP output y. Since the possible reward values were restricted to the set $\{0, 1\}$, an individual could overbid only for instances of the wrong class. A null-initialized set of registers was made available for storing intermediate results and the final output was extracted from a predefined register. Denoting registers by R, inputs by I, and operations by *op*, the instructions themselves could be of the form $R_x \leftarrow op\, R_x\, R_y$ or $R_x \leftarrow op\, R_x\, I_y$. Both unary and binary operators were allowed, and in the cases where *op* was unary it was applied to the y operand.

Five stochastic search operators were used to mutate a parent to generate a child of which the first four affected the bid program: (1) bid delete removed an instruction at an arbitrary position, (2) bid add inserted an arbitrary instruction at a randomly chosen position, (3) bid mutate flipped a random bit of an instruction at an arbitrarily chosen position, (4) bid swap exchanged the positions of two arbitrarily chosen instructions, and (5) action mutate changed the action associated with an individual to a randomly chosen value. Each of these operators was applied with a specified probability and the application of the operators was not exclusive.

During initialization, the individual bid program sizes were selected from a predefined range with uniform probability (i.e., a fixed length representation). The bid delete and add operators were therefore included to allow change to the complexity of a program. The bid swap operator was added for situations when

a program had the right instructions but in the wrong order. Finally, the action mutate operator was employed in case individuals appeared that exhibited the right bidding behaviour but for the wrong class. In addition, this operator proved useful for ensuring an action always had a chance of appearing in the population (e.g., in situations where all individuals advocating an action were extinct).

4 Evaluation and Results

4.1 Datasets and Parameterization

Four real-world datasets from the UCI Machine Learning Repository [12] were used to evaluate the approach, Table 1. The test partitions were generated by randomly selecting instances from the entire dataset to approximately preserve class distributions. With the exception of the BCW dataset where a 50/50 split was used to reduce training times, in all cases one-quarter of all the patterns were held out for testing. BCW refers to the Wisconsin Breast Cancer dataset with patterns containing missing attributes removed. BUPA is the liver disorders databases. Classes 1, 2, and 3 in the Iris dataset correspond to flowers iris-setosa, iris-versicolor, and iris-virginica respectively. Finally, Housing refers to a three-class version of the Boston Housing dataset [9]. The BUPA and Housing datasets are considered to be representative of the more difficult classification problems. It should be noted that individuals in the population were initialized with a single action selected a priori from the set of possible class labels, Table 1, with uniform probability.

Table 1. Datasets used in evaluating the proposed approach. Distribution refers to the number of patterns of each class and is given in the same order as the class labels in the 'Labels' column. All labels are shown as they appear in the original datasets.

			Train		Test	
Dataset	Features	Labels	Patterns	Distribution	Patterns	Distribution
BCW	9	<2, 4>	342	<222, 120>	341	<222, 119>
BUPA	6	<1, 2>	259	<108, 151>	86	<37, 49>
Iris	4	<1, 2, 3>	113	<37, 37, 39>	37	<13, 13, 11>
Housing	13	<1, 2, 3>	380	<123, 140, 117>	126	<44, 33, 49>

The parameters used in all of the experiments are given in Table 2. Thirty different initializations were performed for each configuration to account for the dependence of the algorithm on the starting conditions.

After training, individuals that won zero auctions on the training data were marked as inactive and not used on the test partition. Results were then compiled in terms of the number of *active* individuals, classification accuracy, and bidding behaviour. The results shown are averaged over the thirty initializations performed for each pairing of auction type and dataset.

Table 2. Parameter values used in the experiments

Parameter	Value
Minimum program size	1
Maximum program size	256
Bid delete/add/mutate/swap probability	0.5
Action mutate probability	0.5
Number of registers	4
Function set	$\{+, \times, -, \div, \cos, \sin, \exp, \log\}$
Population size	100
Epochs	100 000
Number of initializations	30

4.2 Results

Figure 1 summarizes the number of active individuals. Compared to the SP auction, the FP auction uses more individuals one the easier datasets and fewer on the difficult ones. Conversely, the SP auction allocates more resources to the more difficult problems and less to the easier problems. Both approaches assign significantly more resources to the more difficult BUPA and Housing datasets.

As seen in the summary of the accuracy results on the test data, Figure 2, there is no clear preference for either auction type. In addition, neither approach is seen to be superior with respect to consistency (i.e., spread of test accuracies).

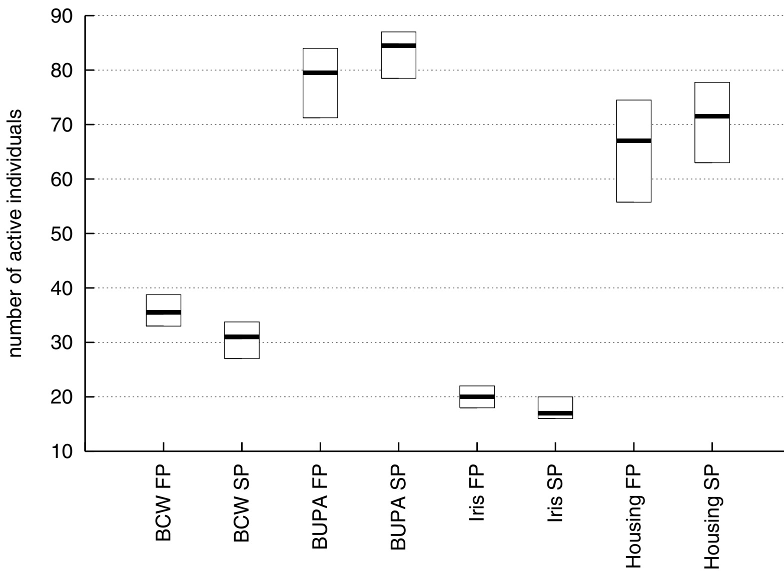


Fig. 1. Active individual counts. The box boundaries denote the first and third quartiles, the thick horizontal line the median.

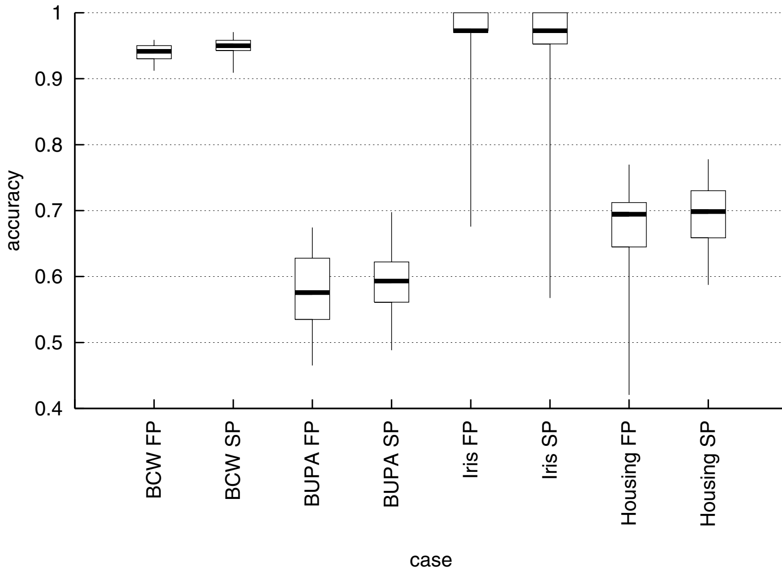
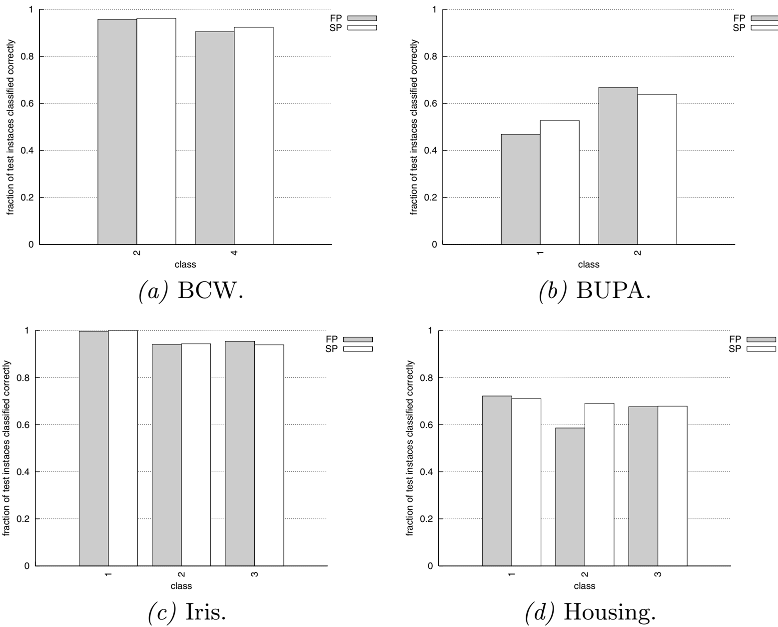


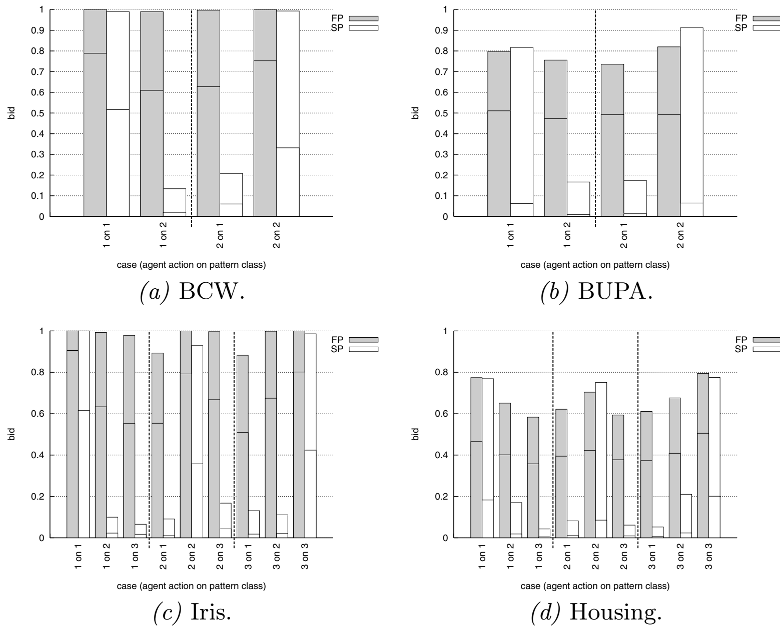
Fig. 2. Accuracy results on the test data. The thick horizontal line denotes the median, the box boundaries the first and third quartiles, and the line endpoints the minimum and maximum.



(a) BCW. *(b)* BUPA.

(c) Iris. *(d)* Housing.

Fig. 3. Per class accuracies on the test data. Each bar shows the accuracy of the FP or SP approach on the class denoted on the x-axis.



(a) BCW. *(b)* BUPA. *(c)* Iris. *(d)* Housing.

Fig. 4. Bidding behaviour on the training data. Each bar shows the maximum mean bid (height of the bar) and the mean bid (segment crossing the area of the bar) for the FP or the SP approach. The bars are grouped by bidding scenarios.

Even though the overall accuracy of the FP and SP approaches appears to be similar, the two differ with respect to their per class accuracies, Figure 3. In the case of the BCW and Iris datasets, the task appears to be straightforward as all classes are represented with a high degree of precision under both auction schemes. On the more difficult BUPA and Housing datasets, however, the SP approach yields more balanced results (i.e., the SP approach is more effective at profiling *all* classes). This suggest that the SP approach will be less biased in situations where the class distributions are unbalanced.

Figure 4 shows aggregate bidding behaviour on the training data. For each case, the mean bid value is calculated by considering the bids of all individuals of a given action on all patterns of a given class. The mean maximum bid is calculated in a similar fashion except that for each pattern only the winning individual is considered (from all individuals advocating a given action). For individuals of action x bidding on instances of class y, the desired behaviour corresponds to bidding high only if $x = y$ and low otherwise. This corresponds to individuals bidding high for patterns that match their actions and low otherwise. Desirable behaviour does not necessarily imply that the mean bid is high whenever $x = y$. Given a class x, certain individuals of action x may bid high only for some of the patterns of class x thus identifying a subclass within a single class (as labeled in the dataset).

Figure 4 shows that the SP approach yields better bidding behaviour. Using the FP approach, the maximum bid always tends to be high; individuals bid high even for patterns associated with the wrong class. This is best illustrated on the BCW dataset where the mean maximum bid using the FP approach is always virtually unity regardless of the true class of the pattern. Using the SP approach, if a pattern does not match the individual's action then that individual's bid tends to be significantly lower. This behaviour appears to be a direct result of the penalty applied to the runner-up in the SP wealth adjustment process and suggests more robust decision boundaries.

4.3 Comparison with C5.0

To summarize and to put the difficulty of the learning task into context, Table 3 shows a comparison of the results obtained using the FP and SP auction types and C5.0 [13]. C5.0 is an established data mining algorithm that can be used to build classifiers in the form of decision trees. In setting up C5.0, all BCW attributes were defined as ordered discrete, the 'CHAS' Housing attribute was defined as discrete, all other attributes were defined as continuous, and default learning parameters were used. The table shows that the proposed approach typically outperforms C5.0.

Table 3. Test accuracies (in percent) of the FP and SP schemes compared to C5.0

	BCW		BUPA		Iris		Housing	
	Best	Mean	Best	Mean	Best	Mean	Best	Mean
FP	96	94	67	58	100	96	77	67
SP	97	95	70	59	100	96	78	70
C5.0	94	-	67	-	97	-	75	-

5 Conclusion

A market-based model for decomposing classification problems between multiple GP individuals was presented. The central mechanism in this model is the auction where an individual can profit only by correctly classifying a problem instance. The proposed approach requires a single population to be evolved and can be directly applied to problems with more than two classes.

Two auction types were examined and the SP formulation found to be superior for several reasons. First, it allocated more resources to the more difficult problems and fewer resources to the easier problems. Second, it yielded more balanced per class classification accuracies. Finally, it produced a wider margin between correct and incorrect bids suggesting more robust decision boundaries.

This work demonstrates that auctions are an effective means of partitioning the instance space in classification problems and that they can be tailored to achieve desired system behaviour. By using bids to associate individuals with patterns, problem decomposition can be achieved. In addition, GP was shown to

be able to successfully evolve the bidding behaviour underlying every auction. In this regard, wealth was shown to be an effective measure of individual fitness.

As demonstrated by Hayek, the idea of using an auction to select an appropriate action given an input is also applicable to reinforcement learning scenarios. One obstacle in this problem domain is the high number of possible test instances. Future work should therefore focus on incorporating active learning to select a manageable and informative subset of test cases during training.

Acknowledgments

The authors would like to thank the Killam Trusts, the Natural Sciences and Engineering Research Council of Canada, and the Canada Foundation for Innovation for their financial support.

References

1. E. B. Baum and I. Durdanovic. An evolutionary post production system. *Advances in Learning Classifier Systems*, pages 3–21, 2000.
2. E. B. Baum and I. Durdanovic. Toward code evolution by artificial economies. *Evolution as Computation*, pages 314–332, 2002.
3. M. Brameier and W. Banzhaf. *Linear Genetic Programming*. Springer, 2007.
4. S. Koenig, C. Tovey, M. Lagoudakis, V. Markakis, D. Kempe, P. Keskinocak, A. Kleywegt, A. Meyerson, and S. Jain. The power of sequential single-item auctions for agent coordination. In *Proceedings of the National Conference on Artificial Intelligence*, 2006.
5. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.
6. I. Kwee, M. Hutter, and J. Schmidhuber. Market-based reinforcement learning in partially observable worlds. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 865–873, 2001.
7. W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.
8. P. L. Lanzi and R. L. Riolo. Recent trends in learning classifier systems research. *Advances in Evolutionary Computing: Theory and Applications*, pages 955–988, 2003.
9. T. Lim, W. Loh, and Y. Shih. A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms. *Machine Learning*, 40:203–228, 2000.
10. A. R. McIntyre and M. I. Heywood. On multi-class classification by way of niching. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 581–592, 2004.
11. A. R. McIntyre and M. I. Heywood. MOGE: GP classification problem decomposition using multi-objective optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 863–870, 2006.
12. D. J. Newman, S. Hettich, C. L. Blake, and C. J. Merz. UCI repository of machine learning databases. [http://www.ics.uci.edu/∼mlearn/mlrepository.html], 1998.
13. Rulequest Research. Data mining tools See5 and C5.0. [http://www.rulequest.com/see5info.html], 2005.
14. S. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

Layered Learning in Boolean GP Problems

David Jackson and Adrian P. Gibbons

Dept. of Computer Science, University of Liverpool,
Liverpool L69 3BX, United Kingdom
d.jackson@csc.liv.ac.uk

Abstract. Layered learning is a decomposition and reuse technique that has proved to be effective in the evolutionary solution of difficult problems. Although previous work has integrated it with genetic programming (GP), much of the application of that research has been in relation to multi-agent systems. In extending this work, we have applied it to more conventional GP problems, specifically those involving Boolean logic. We have identified two approaches which, unlike previous methods, do not require prior understanding of a problem's functional decomposition into sub-goals. Experimentation indicates that although one of the two approaches offers little advantage, the other leads to solution-finding performance significantly surpassing that of both conventional GP systems and those which incorporate automatically defined functions.

1 Introduction

A key obstacle to the more widespread application of evolutionary computing techniques is that although they are often successful in small, clearly-defined domains, they do not always extend well to more complex problems. The issue is often one of scalability, in that evolutionary methods capable of finding solutions to a low-order problem may not prove fruitful when applied to harder versions of the same problem. In an effort to address this, many researchers have investigated ways in which problems may be decomposed into smaller and simpler sub-tasks, the evolved solutions to which can then act as building blocks to solve the original, higher-level problem.

In the field of genetic programming (GP), the most well-known approach to hierarchical evolution is Koza's automatically defined functions (ADFs) [1-3]. In this, the structure of program trees is defined in such a way that each comprises a set of parameterised function branches and a main branch that may invoke those functions. The main branch and the function branches are all subject to the same evolutionary operators, so that they evolve in parallel. Koza and others (e.g. Rosca and Ballard [4]) have provided extensive evidence that problems are often solved more readily by using ADFs than they are without them.

An alternative to the use of ADFs is the technique of Module Acquisition introduced by Angeline and Pollack [5,6]. In this approach, portions of individuals are randomly encapsulated as modules that are protected from the effects of the usual evolutionary operators. Modules are stored in a library, their value being determined by how often they are used by evolving individuals.

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 148–159, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Other approaches are more systematic (i.e. less random) in the modularisation decisions that are made. The Adaptive Representation through Learning (ARL) algorithm [7], for example, identifies and extracts subroutines from offspring which exhibit the best improvements on the fitness of their parents. Similarly, Roberts *et al* [8] describe a two-stage scheme in which module selection is based on sub-tree survival and frequency.

The importance of modularisation has also been recognised for genetic programming systems that use representations which differ from the tree structure conventionally employed. In his work on Cartesian Genetic Programming (CGP), Miller has described bottom-up techniques for building hardware circuits from simpler 'cells' [9], and for encapsulation of modules in evolving CGP programs [10, 11].

In all of the work described above, the emphasis is on *structure*. That is, problem decomposition is enabled by providing environments that allow portions of program structures to be encapsulated as modules which can then be reused. A very different philosophy would be one in which the focus is on *learning*, whereby decomposition is guided by evolutionary pressure to solve pre-specified sub-problems, the genetic material so evolved then forming the foundation for solving the original problem. In a sense, what the system 'learns' from solving the simpler tasks equips it to deal with harder problems. Such a philosophy is embodied in the *layered learning* approach.

In this paper, we investigate the use of layered learning in the context of genetic programming. The work we describe differs from previous research in at least two ways. Firstly, much of the existing work on layered learning has been performed with regard to its application in multi-agent systems. We wish to appraise its use in more conventional GP problems, in particular those involving Boolean logic. Secondly, layered learning as it is conventionally described involves pre-determination of the sub-task skills that should be useful in solving the higher-level problem; i.e. it necessitates some intelligent understanding of the problem domain and how it may be decomposed. By contrast, we aim to explore more mechanistic approaches which do not require such knowledge.

In Section 2 we present a more detailed look at layered learning, and suggest a number of ways in which it may be realised in the context of GP. Sections 3 and 4 describe our experimentation and results for two of these suggested layered learning approaches. These are followed by some conclusions and suggestions for further work.

2 Layered Learning

The layered learning paradigm as described by Stone and Veloso [12] has its roots in earlier work by researchers such as deGaris [13] and Asada *et al* [14]. Intended as a means for dealing with problems for which finding a direct mapping from inputs to outputs is intractable with existing learning algorithms, the essence of the approach is to decompose a problem into subtasks, each of which is then associated with a layer in the problem-solving process. The idea is that the learning achieved at lower layers when solving the simpler tasks directly facilitates the learning required in higher subtask layers. Stone and Veloso presented this paradigm in a domain-independent

general form, and then showed how it could be applied to the specific domain of simulated robotic soccer.

Subsequent research showed how layered learning could be integrated with genetic programming (GP). In particular, the work of Gustafson and Hsu [15-17] has investigated how this combined approach performs when applied in the domain of keep-away soccer, itself a sub-problem of robotic soccer. More recent work has employed the term *incremental reuse using easy missions* rather than layered learning [18].

In layered learning GP (LLGP), evolution begins in a first layer that proceeds towards the solution of a sub-task of the overall problem. The stopping criterion for this may be, say, that a solution for the sub-task is found, or that a certain number of generations has elapsed. The evolutionary process is then ramped up to the next layer, using the genetic material of the previous layer as the initial population to evolve towards the solution of the original high-level problem. Again, there are various ways in which genetic material may be propagated from one layer to the next. The population as it stands at the end of layer one may simply be re-cast as generation zero of the next layer; or the best individuals from the lower layer may be used exclusively to seed the higher layer. It may also be appreciated that the two-layer approach we have outlined could, in principle, be extended to multiple layers.

The use of layered learning necessitates a prior step of decomposing a problem into tasks to be associated with the lower layers. There are at least three ways in which this can be done:

1. Based on the original problem specification, identify a component sub-task.
2. Make use of a subset of the test cases normally used to evaluate the fitness of individuals.
3. Attempt to solve a lower-order form of the same problem.

Much of the work described thus far in the literature adopts the first of these approaches. For example, in the research on keep-away soccer mentioned above, the high-level problem of keeping the opposing team from gaining possession of the ball assumes that it is useful in the lower layer to evolve accurate passing between team-mates without an opposing player present. However, prior decomposition of a problem in this way requires an intelligent understanding of the problem at hand, plus at least some insight into the components that would be useful in forming a solution. For the work we describe here, we are more interested in the other two approaches enumerated above, not only because they have received comparatively little attention to date, but also because the means for realising them is far more mechanical. The next two sections will explore each of these two techniques in turn.

Existing work on layered learning also tends to have a focus in the area of multi-agent systems. We wish to stray from this by concentrating on more 'conventional' GP problems. Again, this seems a largely unexplored area for layered learning, although a related layer-based system has recently been described for solving symbolic regression problems [19]. The focus of our experimentation in the next sections will be on the even-parity and majority-on Boolean GP problems. These have been chosen not only because they are well-known and extensively studied, but also because scalability is a key issue in such problems: although low-order versions are relatively easy to solve, they become rapidly less tractable as the order increases.

3 Layered Learning Using Testing Subsets

In the even-4 parity problem, the aim is to evolve a Boolean design which returns a TRUE output if the number of logic one values on its 4 inputs D0-D3 is even, FALSE otherwise. The function set for the problem is F={AND, OR, NAND, NOR}. In our GP implementation we have used steady state evolution with 5-candidate tournament selection. The population size is 500 and is initialised using the ramped half-and-half approach [1] with no duplicates. The probability of crossover is 0.9, and there is no mutation.

As mentioned in the Introduction, the most common approach to decomposition and reuse in genetic programming is Koza's automatically defined functions (ADFs), and this provides an additional benchmark against which to test the effectiveness of layered learning. In implementing ADFs, we have followed Koza's precept [1] of enabling the evolution of one subroutine for each of the arities from 2 up to n-1, where n is the size of the terminal set. Hence, for the even-4 parity problem, we allow for one subroutine with 2 parameters, and a second with 3 parameters (although not all formal parameters need be used within the body of these functions).

In comparing approaches, we make use of the success rate at finding solutions over 100 runs, each of 50 generations. We also make use of Koza's metric of computational effort [1], defined as the minimum number of individuals that must be processed to achieve a 0.99 probability that a solution will be found. Table 1 compares each of these figures for a standard GP system and an ADF-based GP system in solving the even-4 parity problem.

Table 1. Performance figures for standard GP and ADF-based GP in solving even-4 parity

Approach	Success rate (%)	Comp. Effort
Standard GP	14	700,000
ADF GP	43	97,500

In the first of our approaches to layered learning, we divide the evolutionary process into two layers, the first of which attempts to solve the even-4 parity problem for a subset of the available test cases, the second layer then using this genetic material to work towards solving the problem on all test cases. In moving from one layer to the next, the whole of the population at the end of layer 1 becomes the initial population of the upper layer. No ADFs are involved in the adapted system.

For even-4 parity, exhaustive testing using all combinations of the four inputs {D0, D1, D2, D3} requires 16 test cases. For our first attempt at layered learning, the lower layer will work towards evolving solutions for just 4 of these test cases. Moreover, this subset corresponds to the 4 combinations of values on D0 and D1, so that any solution produced in layer 1 is in fact a solution to the even-2 parity problem on the two inputs {D0, D1}.

One of the first things that experimentation revealed was that the solution of the layer 1 problem requires very little effort. Indeed, programs that pass all 4 test cases were often found in the initial population. The consequence of this is that the initial population for layer 2 was often very similar to that which would have been obtained

had layered learning not been in place. This was reflected in the performance of the layered learning system, which hardly differed from that of the conventional GP system. To combat this, the system was altered so that evolution in layer 1 proceeds until a given saturation level of solutions is obtained in the population. Layer 2 remains unchanged in that it is terminated when a single solution to the even-4 parity problem is found. Table 2 gives the performance of the layered learning GP (LLGP) system for a range of layer 1 solution saturation levels.

Table 2. Performance of LLGP system on even-4 parity, solving for 4 test cases in layer 1

L1 Sat level (%)	**Success rate (%)**	**Comp effort (x1000)**	**L1 Sat level (%)**	**Success rate (%)**	**Comp effort (x1000)**
1	19	440	10	14	560
2	17	486	20	9	975
3	12	637	30	7	1632
4	15	506	40	5	2070
5	15	460	50	6	988

It can be seen from this table that, when layer 1 is allowed to continue until relatively small saturation levels of solutions are obtained (less than 10%) we achieve modest improvements over the conventional GP approach. The best figures occur at the 1% saturation level, where the probability of success is 19% rather than 14%, and the computational effort is 440,000 rather than 700,000. For saturation levels above 10%, the layered learning approach fares rather more poorly than standard GP. Moreover, in comparison to the ADF approach, the performance is markedly worse for all saturation levels.

In a further set of experiments, the amount of work performed in layer 1 was increased, so that instead of finding programs working for just 4 test cases, a total of 8 cases were applied, the aim being to get even closer to a full solution at the end of layer 1. The inputs used covered all combinations of inputs for {D0, D1, D2}, so that solutions in layer 1 corresponded to even-3 parity programs for those inputs. To avoid spending too much time in the lower layer, a cap of 15 generations was placed on it. Table 3 shows the performance results for a range of layer 1 saturations.

Again, the lower saturation percentages resulted in better performance than higher levels, but the figures were nowhere near as good as the ADF results, and were in fact mostly worse than those achieved in the standard GP approach.

Table 3. Performance of LLGP system on even-4 parity, solving for 8 test cases in layer 1

L1 Sat level (%)	**Success rate (%)**	**Comp effort (x1000)**	**L1 Sat level (%)**	**Success rate (%)**	**Comp effort (x1000)**
1	12	814	10	6	1620
2	10	924	20	8	1232
3	14	576	30	6	1462.5
4	12	780	40	7	1200
5	9	1232	50	7	1130

Although the lower layer is generally used to solve a single sub-task, there is no reason why it cannot be used to evolve programs for more than one sub-problem simultaneously. When decomposition is based on a subset of the available test cases, it is possible to have several such subsets, and to work towards evolving given saturation levels of programs for each of those subsets. In the next set of experiments we have done precisely that. Dividing the 16 test cases for even-4 parity into two, we evolve half the population towards solutions for the lower 8 test cases, and the other half of the population towards programs that handle the other 8 test cases. As before, a 15-generation cap is applied. The two halves of the population are then combined to give the initial population for the upper layer. At this stage, it would in principle be possible to bias the selection of mates according to their complementary strengths [20], although we have chosen instead to allow evolution in layer 2 to proceed in the usual way. The results are summarised in Table 4.

Table 4. Performance of LLGP on even-4 parity, with test cases divided into two subsets

L1 Sat level (%)	Success rate (%)	Comp effort (x1000)	L1 Sat level (%)	Success rate (%)	Comp effort (x1000)
1	8	960	10	9	992
2	6	1650	20	7	1632
3	12	560	30	6	1620
4	14	744	40	10	1012
5	10	1029	50	9	1127

Again the results are disappointing. In general, the dual sub-task approach performs worse than the standard GP approach without layered learning, and considerably worse than the ADF-based GP system.

In the majority-on problem, the aim is to evolve a program that is capable of determining whether the majority of its Boolean inputs are set to logic-one. Thus, in the 5-input version, a solution will deliver TRUE if three or more inputs are logic-one, and FALSE otherwise. The function set for the problem is F={AND, OR, NOT}. All other parameters for the problem as we have implemented it are the same as those given for the even-parity problem. In creating the ADF version of the GP code, we have again used Koza's rule of thumb, so that for the majority-5-on problem there are three ADFs, with arities 2, 3 and 4. Table 5 compares the performance of standard GP with the ADF version. As can be seen, an unusual characteristic of this problem is that the performance of the ADF version is substantially worse than that of the conventional GP approach.

Table 5. Performance figures for standard GP and ADF-based GP in solving majority-5-on

Approach	Success rate (%)	Comp. Effort
Standard GP	62	49,000
ADF GP	7	945,000

Table 6 shows the performance results obtained when layered learning is introduced, with layer 1 solving for 8 out of the 32 test input cases. As with even-parity, layered learning with test case subsets seems to offer nothing in the way of improving performance. Only two of the saturation levels (2% and 5%) give a higher probability of success than the standard GP system, but in each case the solutions are found later in the runs, and therefore require greater computational effort.

Table 6. Performance of LLGP system for majority-5-on, solving for 8 test cases in layer 1

L1 Sat level (%)	Success rate (%)	Comp effort (x1000)	L1 Sat level (%)	Success rate (%)	Comp effort (x1000)
1	57	63	10	58	70
2	67	63	20	58	64
3	58	72	30	46	119
4	60	52.5	40	45	119
5	72	63	50	42	117

4 Layered Learning Using Simplified Problems

The second approach to layered learning we wish to consider involves the simplification of problems in such a way that layer 1 attempts to solve a lower-order version of the original problem. In effect, it directly confronts the scalability issue mentioned in the Introduction. For Boolean problems of the nature we are investigating in this paper, such lower-level learning is readily defined. In the case of the even-4 parity problem, evolution in layer 1 can work towards solving either the even-2 or even-3 parity problems; for majority-5-on, layer 1 can be devoted to solving the simpler majority-3-on problem.

A follow-up issue is how best to handle the transfer of genetic material from layer 1 to layer 2, and here we have chosen to take a different path from that used in the previous section. In the experiments described in that section, working solutions to a subset of test cases could not in general be parameterised to make them work for other test subsets. By contrast, when we solve, say, the even-2 parity problem, we evolve a *general* solution that could apply to any pair of inputs. An even-2 parity program that is correct on {D0, D1} can be turned into an even-2 parity program for {D0, D3} simply by replacing all occurrences of D1 in the program by D3. In other words, we can regard layer 1 as solving the even-2 problem for the arbitrary inputs {PAR0, PAR1} where PAR0 and PAR1 can be regarded as formal parameters. This in turn suggests that an appropriate mechanism for layer transfer might be to encapsulate the layer 1 solution as a parameterised module that can be handed on to layer 2 for possible invocation. A further implication is that layer 1 can be terminated as soon as one solution to the lower-order problem is found, since there is little point in trying to evolve multiple modules that are structurally different and yet functionally identical.

In more detail, the layered learning process we propose works like this. The initial population is defined in such a way that each member consists of a single function branch and a main execution branch. In layer 1, evolution focuses solely on the function, with the aim being to evolve a solution to a lower-order version of the problem (e.g. even-2 parity or majority-3-on). As soon as a solution is found, the

resulting function is propagated to all other individuals in the population and we enter layer 2. In this upper layer, evolutionary effort shifts entirely to the main execution branch of the programs, which are free to make use of the function as evolution proceeds towards generating a solution to the higher-order problem (e.g. even-4 parity or majority-5-on).

Table 7 shows the performance results for the even-4 parity problem. The table compares the results obtained from using the layered learning approach we have just outlined with those obtained from both standard GP and GP with ADFs. Two versions of the layered learning system were tried: one in which the lower layer works on the even-2 parity problem, and one in which the lower layer evolves an even-3 parity solution. It should be noted that the computational effort for the layered learning approaches is calculated over *both* layers, and not just the upper layer.

Table 7. Comparison of LLGP with unlayered (monolithic) GP for even-4 parity

Approach	Success rate (%)	Comp. Effort
Monolithic GP	14	700,000
Monolithic GP with ADFs	43	97,500
LLGP using even-2	95	12,000
LLGP using even-3	78	45,000

It is clear that the layered learning approach using simplified problems dramatically out-performs conventional and ADF-based GP. When even-2 parity is used as the sub-problem to be tackled in layer-1, the success rate is almost 7 times greater than that achieved for standard GP, and more than twice as great as that attained for ADF-based GP; moreover, the computational effort is less than 2% of that required for conventional GP, and only about 12% of that required for GP with ADFs.

The success of the approach for even-4 parity encouraged us to try it for the more difficult even-5 parity problem. The only changes to the problem parameters are an additional input D4, and an increase in population size from 500 to 2000. As before, we experimented with versions of the layered learning approach using both even-2 and even-3 parity in the lower layer. The results are compared in Table 8.

Table 8. Comparison of LLGP with monolithic GP for even-5 parity problem

Approach	Success rate (%)	Comp Effort
Monolithic GP	0	-
Monolithic GP with ADFs	32	864,000
LLGP using even-2	92	48,000
LLGP using even-3	64	168,500

Like Koza [1], we found that discovering a solution to the even-5 parity problem using standard GP is extremely difficult. By incorporating an ADF mechanism we were able to get much better results, with a success rate of 32%. Again, however, the

layered learning system using lower-order sub-tasks performs even better. This is particularly so where even-2 parity is solved in layer 1, leading to a success rate that is almost triple that of the ADF-based GP system, and a computational effort ratio of just 5.5%.

The trend continues with higher orders. Table 9 shows the results for the even-6 parity problem, using a population size of 2000. For the ADF-based GP approach, the number of ADFs has been restricted to three. This approach still finds solutions, but is beginning to struggle. As before, both versions of the layered learning approach perform much better.

Table 9. Comparison of LLGP with monolithic GP for even-6 parity problem

Approach	Success rate (%)	Comp Effort
Monolithic GP	0	-
Monolithic GP with ADFs	16	1,056,000
LLGP using even-2	70	120,000
LLGP using even-3	36	570,000

Jumping ahead slightly to the even-8 parity problem, the ADF approach fails to find any solutions when the population size remains at 2000, while the LLGP system which solves for even-2 parity in the lower layer goes on to find even-8 parity solutions at a rate of 25%, and even-9 parity programs with a success rate of 15%.

For the majority-5-on problem, the lower layer was configured to solve the majority-3-on problem. The performance results are given in Table 10. For ease of comparison, the figures for standard GP and ADF-based GP in Table 5 are repeated here.

Table 10. Comparison of LLGP with monolithic GP for majority-5-on problem

Approach	Success rate (%)	Comp Effort
Monolithic GP	62	49,000
Monolithic GP with ADFs	7	945,000
LLGP using maj-3	100	6,000

Again, the performance improvement is dramatic. Using the layered learning approach on simplified problems, every single run evolved a solution to the majority-5-on problem. Moreover, 97% of the solutions were found before generation 7, resulting in a computational effort that is very small in comparison to the other approaches.

As we did with even-parity, we have also tried the approach with a higher order version of the majority-on problem – this time based on 7 inputs, with the lower layer solving majority-3-on. Because of the poor showing of the ADF-based GP system on the majority-5 problem, this method was not attempted again, but Table 11 shows the performance of the layered learning system against standard GP for the majority-7-on problem using a population size of 1000.

Table 11. Comparison of LLGP with monolithic GP for majority-7-on problem

Approach	Success rate (%)	Comp Effort
Monolithic GP	18	1,176,000
LLGP using maj-3	90	54,000

5 Conclusions

In this paper we have investigated the application of layered learning techniques to Boolean logic problems. The two approaches we have tried are both reasonably mechanistic, in that they require no deep *a priori* analysis of the problem's decomposability. The first of these approaches, which involves attempting to solve subsets of the fitness test cases to provide a population base from which to solve the original problem, appears to be disappointing in its effects on genetic programming performance. The second approach, in which solutions to a lower-order version of the problem are solved and encapsulated as modules for use in solving the original problem, seems a lot more promising, and the results yielded by the experiments we performed show impressive performance gains.

There are a number of possible reasons why layered learning with simplified problems performs so well in comparison to the standard ADF approach. Firstly, in a conventional ADF-based system, the architecture of solutions is largely guesswork. It is not known in advance how many functions may be useful, nor how many parameters they require. By contrast, the layered learning approach considers populations in which every individual has precisely one function branch for which the arity is pre-determined by the nature of the sub-task being solved in the lower layer. Secondly, the only evolutionary pressure in an ADF-based GP system is that which pertains to the fitness of the individual as a whole, and any evolutionary changes to functions have to be evaluated within that wider context. In layered learning, on the other hand, evolutionary pressure is applied at a more focused level, with the function itself receiving individual guidance towards a solution. Thirdly, the computational effort in an ADF system may be thinly spread, with the main execution branch and the function-defining branches evolving in parallel. This means that evolutionary operations may be applied to functions which are never invoked or never develop into anything useful, or they may involve changes to the main branch for which evaluation is meaningless since they involve calls to functions which are not yet fully formed. Again, the situation for layered learning is different: all of the computational effort is firstly directed at the single function until it evolves into something potentially useful, and then it is directed exclusively at the main branch.

Despite this success for at least one of the two layered learning approaches, the experimentation we have performed so far has been limited to the domains of the even-parity and majority-on problems. For the future, we plan to investigate its efficacy on other types of problem, including those which are not so obviously scalable (e.g. navigation problems such as the Santa Fe artificial ant trail). We also plan to examine the effects of extending the approach to include multiple functions for handling several sub-tasks, and multiple layers for computationally difficult problems.

References

1. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA (1992)
2. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge, MA (1994)
3. Koza, J.R.: Simultaneous Discovery of Reusable Detectors and Subroutines Using Genetic Programming. In: Proc. 5th International Conf. Genetic Algorithms (ICGA-93) (1993) 295-302
4. Rosca, J.P. and Ballard, D.H.: Hierarchical Self-Organization in Genetic Programming. In: Proc 11th International Conf. on Machine Learning. Morgan Kaufmann, San Francisco, CA (1994) 251-258
5. Angeline, P.J. and Pollack, J.: Evolutionary Module Acquisition. In: Proc. 2nd Annual Conf. on Evolutionary Programming, La Jolla, CA (1993) 154-163
6. Angeline, P.J. and Pollack, J.: Coevolving High-Level Representations. In: Langton, C.G. (ed.): Artificial Life III. Addison-Wesley (1994) 55-71
7. Rosca, J.P. and Ballard, D.H.: Discovery of Subroutines in Genetic Programming. In: Angeline, P. and Kinnear, K.E. Jr. (eds.): Advances in Genetic Programming 2, ch. 9, MIT Press, Cambridge, MA (1996) 177-202
8. Roberts, S.C., Howard, D. and Koza, J.R.: Evolving Modules in Genetic Programming by Subtree Encapsulation. In: Miller, J. et al (eds.): Proc. EuroGP 2001, Lecture Notes in Computer Science, vol. 2038, Springer-Verlag, Berlin Heidelberg (2001) 160-175
9. Miller, J.F. and Thomson, P.: A Developmental Method for Growing Graphs and Circuits. In: Proc. 5th International Conf. on Evolvable Systems, Trondheim, Norway (2003) 93-104
10. Walker, J.A. and Miller, J.F.: Evolution and Acquisition of Modules in Cartesian Genetic Programming. In: Keijzer, M. et al (eds.): Proc. EuroGP 2004, Lecture Notes in Computer Science, vol. 3003, Springer-Verlag, Berlin Heidelberg (2004) 187-197
11. Walker, J.A. and Miller, J.F.: Improving the Performance of Module Acquisition in Cartesian Genetic Programming. In: Beyer, H-G. and O'Reilly, U-M. (eds.): Proc. GECCO 2005, ACM Press, New York (2005) 1649-1656
12. Stone, P. and Veloso, M.: Layered Learning. In: Proc. 17th International Conf. on Machine Learning, Springer-Verlag, Berlin Heidelberg (2000) 369-381
13. deGaris, H.: Genetic Programming: Building Artificial Nervous Systems Using Genetically Programmed Neural Network Modules. In Porter, B.W. et al (eds.): Proc. Seventh International Conf. on Machine Learning (ICML-90) (1990) 132-139
14. Asada, M., Noda, S., Tawaratsumida, S. and Hosoda, K.: Purposive Behaviour Acquisition for a Real Robot by Vision-Based Reinforcement Learning. In: Machine Learning, vol. 23 (1998) 279-303
15. Gustafon, S.M.: Layered Learning in Genetic Programming for a Cooperative Robot Soccer Problem. M.S. Thesis, Dept. of Computing and Information Sciences, Kansas State University, USA (2000)
16. Gustafon, S.M. and Hsu, W.H.: Layered Learning in Genetic Programming for a Cooperative Robot Soccer Problem. In: Miller, J.F. et al (eds.): Proc. EuroGP 2001, Lecture Notes in Computer Science, vol. 2038, Springer-Verlag, Berlin Heidelberg (2001) 291-301
17. Hsu, W.H. and Gustafon, S.M: Genetic Programming and Multi-Agent Layered Learning by Reinforcements. In: Proc. GECCO 2002, New York, NY, USA (2002) 764-771

18. Hsu, W.H., Harmon, S.J., Rodriguez, E. and Zhong, C.: Empirical Comparison of Incremental Reuse Strategies in Genetic Programming for Keep-Away Soccer. In: GECCO 2004 late-breaking papers (2004)
19. Tuan-Hao, H., McKay, R.I., Essam, D. and Hoai, N.X.: Solving Symbolic Regression Problems Using Incremental Evaluation in Genetic Programming. In: Proc. 2006 IEEE Congress on Evolutionary Computation (CEC), Vancouver, BC, Canada (2006) 7487-7494
20. Dolin, B., Arenas, M.G. and Merelo, J.J.: Opposites Attract: Complementary Phenotype Selection for Crossover in Genetic Programming. In: Merelo Guervas, J.J. et al (eds.): PPSN VII, Lecture Notes in Computer Science, vol. 2439, Springer-Verlag, Berlin Heidelberg (2002) 142-152

Mining Distributed Evolving Data Streams Using Fractal GP Ensembles

Gianluigi Folino, Clara Pizzuti, and Giandomenico Spezzano

Institute for High Performance Computing and Networking, CNR-ICAR
Via P. Bucci 41C
87036 Rende (CS), Italy
{folino,pizzuti,spezzano}@icar.cnr.it

Abstract. A Genetic Programming based boosting ensemble method for the classification of distributed streaming data is proposed. The approach handles flows of data coming from multiple locations by building a global model obtained by the aggregation of the local models coming from each node. A main characteristics of the algorithm presented is its adaptability in presence of concept drift. Changes in data can cause serious deterioration of the ensemble performance. Our approach is able to discover changes by adopting a strategy based on self-similarity of the ensemble behavior, measured by its fractal dimension, and to revise itself by promptly restoring classification accuracy. Experimental results on a synthetic data set show the validity of the approach in maintaining an accurate and up-to-date GP ensemble.

1 Introduction

Ensemble learning algorithms [1,5,2,8] based on Genetic Programming (GP) [11,16,12,3,7] have been gathering an increasing interest in the research community because of the improvements that *GP* obtains when enriched with these methods. These approaches have been applied to many real world problems and assume that all training data is available at once. However, in the last few years, many organizations are collecting a tremendous amount of data that arrives in the form of continuous stream. Credit card transactional flows, telephone records, sensor network data, network event logs are just some examples of streaming data. Processing these kind of data poses two main challenges to existing data mining methods. The first is relative to the performance and the second to adaptability.

Many data stream algorithms have been developed over the last decade for processing and mining data streams that arrive at a single location or at multiple locations. Some of these algorithms, known as centralized data stream mining (CDSM) algorithms, require that the data be sent to one single location before processing. These algorithms, however, are not applicable in cases where the data, computation, and other resources are distributed and cannot or should not be centralized for a variety of reasons e.g. low bandwidth, security, privacy issues, and load balancing. In many cases the cost of centralizing the data can

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 160–169, 2007.
© Springer-Verlag Berlin Heidelberg 2007

be prohibitive and the owners may have privacy constraints. Unlike the traditional centralized systems, the distributed data mining (DDM) systems offer a fundamental distributed solution to analyze data without necessarily demanding collection of the data to a single central site. Typically DDM algorithms involve local data analysis to extract knowledge structures represented in models and patterns and the generation of a global model through the aggregation of the local results.

The ensemble paradigm is particularly suitable to support the DDM model. However, to extract knowledge from streaming information the ensemble must adapt its behavior to changes that occur into the data over time.

Incremental or online methods [9,18] are an approach able to support adaptive ensembles on evolving data streams. These methods build a single model that represents the entire data stream and continuously refine their model as data flows. However, maintaining a unique up-to-date model might preclude valuable information to be used since previously trained classifiers have been discarded. Furthermore, incremental methods are not able to capture new trends in the stream. In fact, traditional algorithms assume that data is static, i.e. a concept, represented by a set of features, does not change because of modifications of the external environment. In the above mentioned applications, instead, a concept may drift due to several motivations, for example sensor failures, increases of telephone or network traffic. Concept drift can cause serious deterioration of the ensemble performance and thus its detection allows to design an ensemble that is able to revise itself and promptly restore its classification accuracy.

Another approach to mine evolving data streams is to capture changes in data by measuring online accuracy deviation over time and deciding to recompute the ensemble if the deviation has exceeded a pre-specified threshold. These methods are more effective and allow to handle the concept drift problem in order to capture time-evolving trends and patterns in the stream.

In this paper we a propose a distributed data stream mining approach based on the adoption of an ensemble learning method to aggregate models trained on distributed nodes, and enriched with a change detection strategy to reveal changes in evolving data streams. We present an adaptive GP boosting ensemble algorithm for classifying data streams that maintains an accurate and up-to-date ensemble of classifiers for continuous flows of data with concept drifts. The algorithm uses a DDM approach where not only data is distributed, but also the data is non-stationary and arriving in the form of multiple streams. The method is efficient since each node of the network works with its local data, and communicate the local model computed with the other peer-nodes to obtain the results. A main characteristics of the algorithm is its ability to discover changes by adopting a strategy based on self-similarity of the ensemble behavior, measured by its fractal dimension, and to revise itself by promptly restoring classification accuracy. Experimental results on a synthetic data set show the validity of the approach in maintaining an accurate and up-to-date GP ensemble.

The paper is organized as follows. The next section recall the ensemble technique. Section 3 presents the adaptive GP ensemble method, and the technique proposed for change detection. In section 4, finally, the results of the method on a synthetic data set are presented.

2 Ensemble for Streaming Data

An ensemble of classifiers is constituted by a set of predictors that, instead of yielding their individual decisions to classify new examples, combine them together by adopting some strategy [2,8,5,1]. Boosting is an ensemble technique introduced by Schapire and Freund [8] for boosting the performance of any "weak" learning algorithm, i.e. an algorithm that "generates classifiers which need only be a little bit better than random guessing". The boosting algorithm, called *AdaBoost*, adaptively changes the distribution of the training set depending on how difficult each example is to classify. Given the number T of trials (rounds) to execute, T weighted training sets $S_1, S_2, \ldots, S_T$ are sequentially generated and T classifiers $C^1, \ldots, C^T$ are built to compute a weak hypothesis h_t. Let w_i^t denote the weight of the example x_i at trial t. At the beginning $w_i^1 = 1/n$ for each x_i. At each round $t = 1, \ldots, T$, a weak learner C^t, whose error ϵ^t is bounded to a value strictly less than 1/2, is built and the weights of the next trial are obtained by multiplying the weight of the correctly classified examples by $\beta^t = \epsilon^t/(1-\epsilon^t)$ and renormalizing the weights so that $\Sigma_i w_i^{t+1} = 1$. Thus "easy" examples get a lower weight, while "hard" examples, that tend to be misclassified, get higher weights.

In the last few years many approaches to processing data streams through classifier ensembles have been proposed. Street and Kim [17] build individual classifiers from chunks of data read sequentially in blocks. They are then combined into an ensemble of fixed size. When the ensemble is full, new classifiers are added only if they improve the ensemble's performance. Concept drift is treated by relying on the replacement policy of the method. Wang et al. [19] propose a framework for mining concept drifting data streams using weighted ensemble classifiers. The classifiers are weighted by estimating the expected prediction error on the test set. The size K of the ensemble is maintained constant by considering after each block of data the first top K weighted classifiers. Chu and Zaniolo [4] present a boosting algorithm modified to classify data streams able to handle concept drift via change detection. The boosting algorithm trains a new classifier on a data block whose instances are weighted by the ensemble built so far. Changes are discovered by modelling the ensemble accuracy as a random variable and performing a statistical test. When a change is detected the weights of the classifiers are reset to 1 and the boosting algorithm restarts. The ensemble is updated by substituting the oldest predictor with the last created.

As regards Genetic Programming, to the best of our knowledge, there is not any approach in the literature that cope with the extension of GP ensemble learning techniques to deal with streaming data. In the next section our adaptive GP boosting ensemble method is described.

3 Adaptive GP Boosting Ensemble

In this section the description of the algorithm *StreamGP* is given. The method builds an ensemble of classifiers by using, at every round of the boosting procedure, the algorithm *CGPC* [6] on each node to create a population of predictors. The ensemble is then used to predict the class membership of new streams of data and updated only when concept drift is detected. This behavior has a twofold advantage. The first is that it saves a lot of computation because the boosting algorithm is executed only if there is a significant deterioration of the ensemble performance. The second is that the ensemble is able to promptly adapt to changes and restore ensemble accuracy. Change identification is handled at every block. This means that each data block is scanned at most twice. The first time the ensemble predicts the class label of the examples contained in that block. The second scan is executed only if the ensemble accuracy on that block is below the value obtained so far. In such a case, in fact, the boosting algorithm is executed to obtain a new set of classifiers to update the ensemble.

3.1 *StreamGP*

StreamGP is an adaptive GP boosting ensemble algorithm for classifying data streams that applies the boosting technique in a distributed hybrid multi-island model of parallel GP. The hybrid model modifies the multi-island model by substituting the standard GP algorithm with a cellular GP algorithm. In the cellular model each individual has a spatial location, a small neighborhood and interacts only within its neighborhood. In our model we use the *CGPC* algorithm in each island. *CGPC* generates a classifier as a decision tree where the function set is the set of attribute tests and the terminal set are the classes. When a tuple has to be evaluated, the function at the root of the tree tests the corresponding attribute and then executes the argument that outcomes from the test. If the argument is a terminal, then the class name for that tuple is returned, otherwise the new function is executed. CGPC generates a classifier as follows. At the beginning, for each cell, the fitness of each individual is evaluated. The fitness is the number of training examples classified in the correct class. Then, at each generation, every tree undergoes one of the genetic operators (reproduction, crossover, mutation) depending on the probability test. If crossover is applied, the mate of the current individual is selected as the neighbor having the best fitness, and the offspring is generated. The current tree is then replaced by the best of the two offsprings if the fitness of the latter is better than that of the former. After the execution of the number of generations defined by the user, the individual with the best fitness represents the classifier.

The boosting schema is extended to cope with continuous flows of data and concept drift. Let M be the fixed size of the ensemble $E = \{C_1, \ldots, C_M\}$. Once the ensemble has been built, by running the boosting method on a number of blocks, the main aim of the adaptive *StreamGP* is to avoid to train new classifiers as new data flows in until the performance of E does not deteriorate very much, i.e. the ensemble accuracy maintains above an acceptable value.

```
     Given a network constituted by p nodes, each having a streaming data set
1.   E = ∅
2.   F = ∅
3.   for j = 1 ... p  (each island in parallel)
4.     while (more_Blocks)
5.       Given a new block B_k = {(x_1, y_1), ... (x_n, y_n)}, x_i ∈ X
         with labels y_i ∈ Y = {1, 2, ..., d}
6.       evaluate the ensemble E on B_k and let f_k be the fitness value obtained
7.       if | F |< H
8.         F = F ∪ f_k
9.       else F = {F − {f_1}} ∪ f_k
10.      compute the fractal dimension F_d of the set F
11.      if (F_d(F) < τ)
12.        Initialize the subpopulation Q_i with random individuals
13.        Initialize the example weights w_i = 1/n for i = 1, ..., n
14.        for t = 1, 2, 3, ..., T (for each round of boosting)
15.          Train CGPC on the block B_k using a weighted fitness
             according to the distribution w_i
16.          Learn a new classifier C_t^j
17.          Exchange the p classifiers C_t^1, ..., C_t^p obtained among the p processors
18.          Update the weights
19.          E = E ∪ {C_t^1, ..., C_1^p }
20.        end for
21.        if (| E |> M) prune the ensemble E
22.      end if
23.    end while
24.  end parallel for
```

Fig. 1. The StreamGP algorithm

To this end, as data comes in, the ensemble prediction is evaluated on these new chunks of data, and augmented misclassification errors, due to changes in data, are detected by using the notion of *fractal dimension*, described in the next section, to the set $F = \{f_1, \ldots, f_H\}$ containing the last H fitness values obtained by evaluating the ensemble on the blocks. When an alarm of change is revealed, the GP boosting schema generates new classifiers, thus a decision on which classifiers must be discarded from the ensemble, because no longer consistent with the current concepts, has to be done. A simple technique, often adopted in many existing method, and also in our approach, is to eliminate the older predictors and substitute them with the most recent ones.

The description of the algorithm in pseudo-code is shown in figure 1. Let a network of p nodes be given, each having a streaming data set. As data continuously flows in, it is broken in blocks of the same size n. Every time a new block B_k of data is scanned, the ensemble E obtained so far is evaluated on B_k and the fitness value obtained f_k is stored in the set F (steps 5-6). $F = \{f_1, \ldots, f_H\}$ contains the last H evaluations of E on the data blocks, that is the fitness value set on which the fractal dimension $F_d(F)$ is computed (step 10). If $F_d(F)$ is

below a fixed threshold value τ (step 11), then it means that a change in data is detected, thus the ensemble must adapt to these changes by retraining on the new data set. To this end the boosting standard method is executed for a number T of rounds (steps 12-20). For every node N_i, $i = 1, \ldots, p$ of the network, a subpopulation Q_i is initialized with random individuals (step 12) and the weights of the training instances are set to $1/n$, where n is the data block size (step 13). Each subpopulation Q_i is evolved for T generations and trained on its local block B_k by running a copy of the $CGPC$ algorithm (step 15). Then the p individuals of each subpopulation (step 16) are exchanged among the p nodes and constitute the ensemble of predictors used to determine the weights of the examples for the next round (steps 17-19). If the size of the ensemble is more than the maximum fixed size M, the ensemble is pruned by retiring the oldest $T \times p$ predictors and adding the new generated ones (step 21).

3.2 Change Detection

An important step of the above described algorithm is the detection of changes in the data distribution that causes significant deterioration of the ensemble accuracy. In this section we propose to use the notion of *fractal dimension* to discover concept drift in streaming data. *Fractals* [14] are particular structures that present *self-similarity*, i. e. an invariance with respect to the scale used. The fractal dimension of fractal sets can be computed by embedding the data set in a d-dimensional grid whose cells have size r and computing the frequency p_i with which data points fall in the i-th cell. The fractal dimension D [10] is given by the formula $D = \frac{1}{q-1}\frac{log \sum_i p_i^q}{log\ r}$. Among the fractal dimensions, the *correlation dimension*, obtained when $q = 2$ measures the probability that two points chosen at random will be within a certain distance of each other. Changes in the correlation dimension mean changes in the distribution of data in the data set, thus it can be used as an indicator of concept drift. Fast algorithms exist to compute the fractal dimension. We applied the $FD3$ algorithm of [15] that implements the *box counting method* [13]. In order to employ the fractal dimension concept to our approach, we proceed as follows. Suppose we have already scanned k blocks $B_1, \ldots, B_k$ and computed the fitness values $\{f_1, \ldots, f_k\}$ of the ensemble on each block. Let $F = \{f_1, \ldots, f_H\}$ be the fitness values computed on the most recent H blocks, and $F_d(F)$ be the fractal dimension of F. When the block B_{k+1} is examined, let f_{k+1} be the fitness value of the GP ensemble on it. If $F_d(F \cup \{f_{k+1}\}) < \tau$, where τ is a fixed threshold, then the fractal dimension shows a decrease. This means that data distribution has been changed and the ensemble classification accuracy drops down. This approach has been shown to be very effective experimentally. In the next section we show that when the misclassification error of the ensemble increases, the fractal dimension drops down.

4 Experimental Results

In this section we study the effectiveness of our approach on a synthetic data set with two classes introduced in [4]. Geometrically the data set is a 5-dimensional

unit hypercube, thus an example x is a vector of 5 features $x_i \in [0, 1]$. The class boundary is a hyper-sphere of radius r and center c. If an example x is inside the sphere then it is labelled class 1, class 0 otherwise. Furthermore, the data set contains a noise level of 5% obtained by flipping randomly the class of the tuples from 0 to 1 and viceversa, with probability 0.05.

The experiments were performed using a network composed by 5 1.133 Ghz Pentium III nodes having 2 Gbytes of Memory, interconnected over high-speed LAN connections. On each node a data set consisting of 360k tuples was generated by using as center the point $(0.5, 0.5, 0.5, 0.5, 0.5)$ and radius 0.25. The algorithm receives blocks of size 1k. Every 40k tuples the data set is perturbed by moving the center of 0.15 in a randomly chosen direction. This generates the migration of many points from class 0 to class 1 and viceversa. Thus, at blocks 40, 80, 120, 160 and so, i.e. each 40 blocks, concept drift is forced by provoking a deterioration of the ensemble performance that should be restored by our algorithm.

On each node a population of 100 predictors is evolved with a probability of 0.1 for reproduction, 0.8 for crossover and 0.1 for mutation. The maximum depth of the new generated subtrees is 4 for the step of population initialization, 17 for crossover and 2 for mutation. We used T=5 rounds for the boosting, each round executing for 100 generations. Thus *CGPC* is trained on each block for 500 generations. At the end of the 5 rounds, each node produces 5 predictors, one for round, thus, since we have 5 nodes, 25 new classifiers are generated every time the fractal dimension diminishes below the threshold τ. This means that the oldest 25 classifiers are substituted by these new ones.

The experiments aimed at evaluating the ability of the approach in discovering concept drift and in restoring classification accuracy. Figure 2 reports the fitness, i.e the classification accuracy, and the value of the fractal dimension respectively when an ensemble of size 100 (on the left) and 200 (on the right) are used. The figure points out the abrupt deterioration of classification accuracy every 40 blocks of data (solid lines) and the corresponding decrease of the fractal dimension (dotted lines), thus allowing to reveal the change and to retrain the ensemble on the new block. Figures 2(a),(c),(e),(g) and (b),(d),(f),(h) show the effect of different values of the threshold τ, i.e. 0.80, 0.85, 0.87, 1.0, when the ensemble size is 100 and 200 respectively. The horizontal solid line on each figure indicates the τ value. The figures point out that, independently the value of τ, a larger ensemble is able to restore classification accuracy more quickly. Furthermore, higher the value of τ, faster the detection of change and more quickly the ensemble performance fixed up. In fact, every time the fractal dimension is below τ, the algorithm executes the boosting method, steps 12-21 in figure 1, and updates the ensemble with the new classifiers obtained by retraining *CGPC* on the new data set. If $\tau = 1$ the retraining is executed at each block, that is it is assumed that changes occur at each new data block. Thus a higher value of τ implies a heavier computational load for the algorithm but a better classification accuracy. Its choice must take into account the trade-off between these two factors. In table 1 the percentage of blocks on which the boosting phase is executed

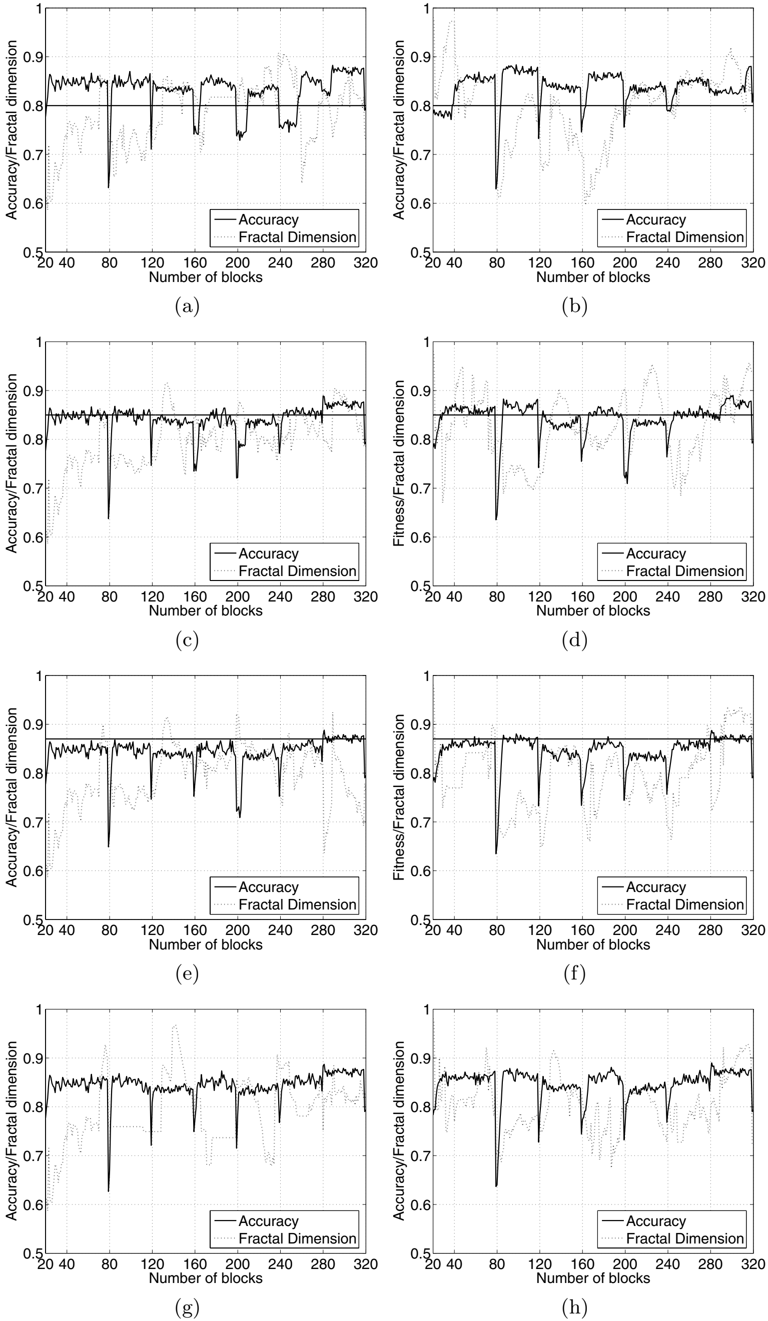


Fig. 2. Fitness (solid lines) and Fractal Dimension (dotted lines) with ensemble size 100, on the left, and 200 on the right, for different thresholds of F_d (a),(b): 0.80, (c),(d): 0.85, (e), (f): 0.87 and (g), (h): 1.0

and the corresponding mean classification accuracy are reported. For example, a 0.80 value for τ permits to save about 50% of computation, maintaining a good accuracy. An ensemble of 200 classifiers ulteriorly reduces the computational load since the retraining is executed for nearly the 35% of blocks. It is worth to note that the gain in accuracy when the boosting method is executed for each new block (last row of table 1) is marginal. For example, an ensemble of 200 predictors that is retrained at each block obtains an accuracy of 84.78, while if it is retrained on about the half of blocks, the accuracy is 84.37. This result substantiate the validity of the approach proposed.

Table 1. Percentage of blocks used in the training phase for different values of fractal threshold, and average classification accuracy

	100 classifiers		200 classifiers	
τ	Blocks	Accuracy	Blocks	Accuracy
0.80	47.21 %	83.28 %	34.90 %	83.73 %
0.85	82.72 %	84.18 %	54.84 %	84.37 %
0.87	93.69 %	84.36 %	82.11 %	84.69 %
1	100.0%	84.42%	100.0%	84.78 %

5 Conclusions

The paper presented a GP boosting ensemble method for the classification of distributed streaming data that comes from multiple locations. The method is able to handle concept drift via change detection. Changes are discovered by adopting a strategy based on self-similarity of the ensemble behavior, measured by its fractal dimension. This allows the ensemble to revise itself and promptly restore classification accuracy. Experimental results on a synthetic data set showed the validity of the approach in maintaining an accurate and up-to-date GP ensemble. Future work aims at studying parameter tuning and to test the approach on real streaming data sets.

Acknowledgements. This work has been partially supported by the LOGICA project funded by Regione Calabria (Programma Operativo Regionale POR, Misura 3.16.B2).

References

1. Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, (36):105–139, 1999.
2. Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.
3. E. Cantú-Paz and C. Kamath. Inducing oblique decision trees with evolutionary algorithms. *IEEE Transaction on Evolutionary Computation*, 7(1):54–68, February 2003.

4. Fang Chu and Carlo Zaniolo. Fast and light boosting for adaptive mining of data streams. In Honghua Dai, Ramakrishnan Srikant, and Chengqi Zhang, editors, *Proceedings of the 8th Pacific-Asia Conference (PAKDD 2004), May 26-28, 2004, Proceedings*, volume 3056 of *LNAI*, pages 282–292, Sydney, Australia, 2004. Springer Verlag.
5. Thomas G. Dietterich. An experimental comparison of three methods for costructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine Learning*, (40):139–157, 2000.
6. G. Folino, C. Pizzuti, and G. Spezzano. A cellular genetic programming approach to classification. In *Proc. Of the Genetic and Evolutionary Computation Conference GECCO99*, pages 1015–1020, Orlando, Florida, July 1999. Morgan Kaufmann.
7. G. Folino, C. Pizzuti, and G. Spezzano. A scalable cellular implementation of parallel genetic programming. *IEEE Transaction on Evolutionary Computation*, 10(5):604–616, October 2006.
8. Y. Freund and R. Scapire. Experiments with a new boosting algorithm. In *Proceedings of the 13th Int. Conference on Machine Learning*, pages 148–156, 1996.
9. J. Gehrke, V. Ganti, R. Ramakrishnan, and W. Loh. Boat - optimistic decision tree construction. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'99)*, pages 169–180. ACM Press, 1999.
10. P. Grassberger. Generalized dimensions of strange attractors. *Physics Letters*, 97A:227–230, 1983.
11. Hitoshi Iba. Bagging, boosting, and bloating in genetic programming. In *Proc. Of the Genetic and Evolutionary Computation Conference GECCO99*, pages 1053–1060, Orlando, Florida, July 1999. Morgan Kaufmann.
12. W.B. Langdon and B.F. Buxton. Genetic programming for combining classifiers. In *Proc. Of the Genetic and Evolutionary Computation Conference GECCO'2001*, pages 66–73, San Francisco, CA, July 2001. Morgan Kaufmann.
13. L. Liebovitch and T. Toth. A fast algorithm to determine fractal dimensions by box counting. *Physics Letters*, 141A(8):–, 1989.
14. B. Mandelbrot. *The Fractal Geometry of Nature*. W.H Freeman, New York, 1983.
15. J. Sarraille and P. DiFalco. *FD3*. http://tori.postech.ac.kr/softwares.
16. Terence Soule. Voting teams: A cooperative approach to non-typical problems using genetic programming. In *Proc. Of the Genetic and Evolutionary Computation Conference GECCO99*, pages 916–922, Orlando, Florida, July 1999. Morgan Kaufmann.
17. W. Nick Street and YongSeog Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the seventh ACM SIGKDD International conference on Knowledge discovery and data mining (KDD'01),*, pages 377–382, San Francisco, CA, USA, August 26-29, 2001 2001. ACM.
18. P. E. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4:161–186, 1989.
19. H. Wang, Wei Fan, P.S. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the nineth ACM SIGKDD International conference on Knowledge discovery and data mining (KDD'03),*, pages 226–235, Washington, DC, USA, August 24-27, 2003 2003. ACM.

Multi-objective Genetic Programming for Improving the Performance of TCP

Cyril Fillon and Alberto Bartoli

University of Trieste, Via Valerio, 10, 34127 Trieste, Italy
{cfillon,bartolia}@univ.trieste.it

Abstract. TCP is one of the fundamental components of the Internet. The performance of TCP is heavily dependent on the quality of its *round-trip time (RTT) estimator*, i.e. the formula that predicts dynamically the delay experienced by packets along a network connection. In this paper we apply *multi-objective genetic programming* for constructing an RTT estimator. We used two different approaches for multi-objective optimization and a collection of real traces collected at the mail server of our University. The solutions that we found outperform the RTT estimator currently used by all TCP implementations. This result could lead to several applications of genetic programming in the networking field.

1 Introduction

Genetic programming is a powerful framework for coping with problems in which finding a solution is difficult but evaluating the performance of a candidate solution is reasonably simple [12,5]. Many engineering problems exhibit this feature and may thus greatly benefit from genetic programming techniques. Real-world engineering problems, on the other hand, can only be solved based on a trade-off amongst multiple and often conflicting performance objectives. Several approaches for such *multi-objective optimization* problems have been proposed in evolutionary computing [7], mostly for genetic algorithms [10,16,15] and more recently also for genetic programming [9,14].

In this paper we apply two techniques for multi-objective optimization in genetic programming to an important real-world problem in the Internet domain: the construction of a *round-trip time (RTT) estimator* for TCP [11,4,13]. TCP (Transmission Control Protocol) is a fundamental component of the Internet as it constitutes the basis for many applications of utmost importance, including the World Wide Web and the e-mail just to mention only the most widely known. The TCP implementation internally maintains a dynamic estimator of the round-trip time, i.e., the time it takes for a packet to reach the other endpoint of a connection and to come back. This component has a crucial importance on the overall TCP performance [11,4]. The construction of an RTT estimator is a particularly challenging problem for genetic programming because, as we shall see in more detail, an RTT estimator must satisfy two conflicting requirements, there are many solutions that are optimal for only one of the two requirements and any such solution performs poorly for the other one.

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 170–180, 2007.
© Springer-Verlag Berlin Heidelberg 2007

We construct RTT estimators via multi-objective genetic programming and evaluate their performance on real traces collected at the mail server of our University. The formulas that we found outperform the RTT estimator used in all existing TCP implementations [11] — including those in Windows 2000/XP, Linux, Solaris and so on. We believe this result is particularly significant and could lead to several interesting developments and applications of genetic programming in the networking field.

The outline of the paper is as follows. The next section presents in a first part the RTT estimation problem in detail and introduces the fundamental concepts and techniques in multi-objectives optimization in a second part. Section 3 describes several multi-objective strategies used on our problem. Section 4 presents the experimental procedure used to discover new formulas which estimate RTT. Section 5 discusses the behavior of the different multi-objectives policies as well as the performances of the formula found by Genetic Programming. Finally conclusions are drawn.

2 RTT Estimation Problem

The Transmission Control Protocol (TCP) provides a transport layer, base of many other protocols used in the most common Internet applications, like HTTP (i.e., the Web), FTP (files transfer) and SMTP/POP3 (e-mail). TCP was defined in [1,2]. We provide in the following only the necessary background for this work. More details on the TCP implementation can be found in many places, for example, in [3].

TCP provides applications with a *reliable* and *connection-oriented* service. This means that two remote applications can establish a connection between

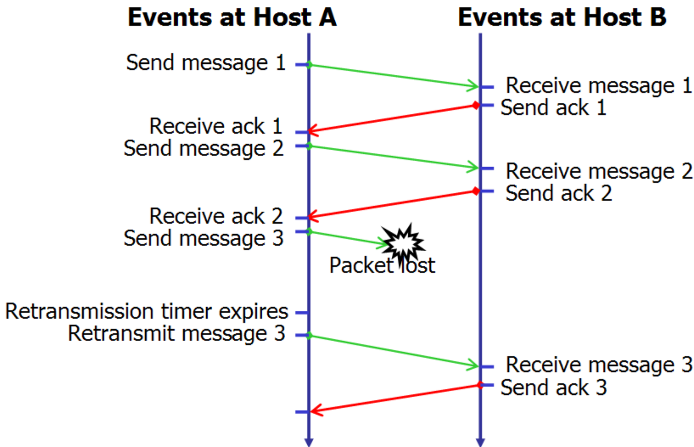


Fig. 1. An example of retransmission for the case (i)

them and that bytes inserted at one end will reach the other end reliably, that is, without losses, in order and without duplicates.

To ensure reliable delivery of packets in spite of packet losses, that may occur at lower levels of the Internet protocol stack, the TCP implementation employs internally a retransmission scheme based on *acknowledgments* as follows. Consider a connection between hosts A and B. Whenever either of them, say A, sends a packet S to the other, it sets a *retransmission timeout (RTO)*. Whenever B receives a packet S, it responds with another packet $ack(S)$ for notifying the other end that S has been indeed received. If A does not receive $ack(S)$ before RTO expires, then A resends S. Note that when RTO expires only one of the following is true, but A cannot tell which one: (i) S has been lost; (ii) S was received by B but $ack(S)$ is lost; (iii) neither S nor $ack(S)$ was lost and RTO expired too early. The fact that A resends S whenever RTO expires means that the TCP implementation *assumes* that case (i) always holds. The three cases described above are illustrated in Figure 1 and 2.

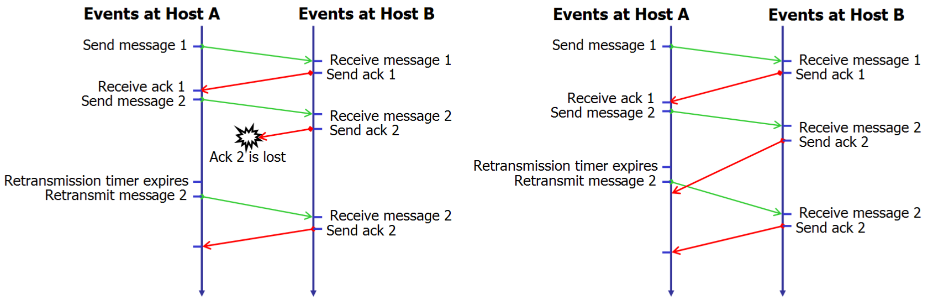


Fig. 2. Examples of retransmission for case (ii) and (iii)

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

RTT varies dynamically, due to the varying delays experienced by packets along the route to their destination. Moreover, when sending packet S_i the corresponding RTT value $measuredRTT_i$ is not yet known. The TCP implementation thus maintains dynamically, on a per-connection basis, an *estimated RTT* and selects RTO for S_i based on the current value for $estimatedRTT_i$. This component of TCP has a crucial importance on performance of TCP [4].

Virtually *all* the TCP implementations – including those in Linux, Windows 2000/XP, Solaris and so on – maintain $estimatedRTT_i$ according to an algorithm

due to Jacobson [11]. This algorithm constructs $estimatedRTT_i$ based on the previous estimate $estimatedRTT_{i-1}$ and the previous value actually observed $measuredRTT_{i-1}$:

$$estimatedRTT_i = (1 - k_1)\, estimatedRTT_{i-1} + k_1\, measuredRTT_{i-1} \quad (1)$$

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

In this work we are concerned with RTT estimation only, i.e., we seek for methods for estimating RTT that are different from formula 1 and hopefully better. The construction of $estimatedRTT$ has two conflicting objectives to optimize. One would like to minimize the number of underestimates (which may cause premature timeout expiration) while at the same time minimizing the average error (which may cause excessive delay when reacting to a packet loss). The problem is challenging because optimizing the former objective is very simple — any very large estimation would work fine — but many excellent solutions from that point of view are very poor from the point of view of the average error.

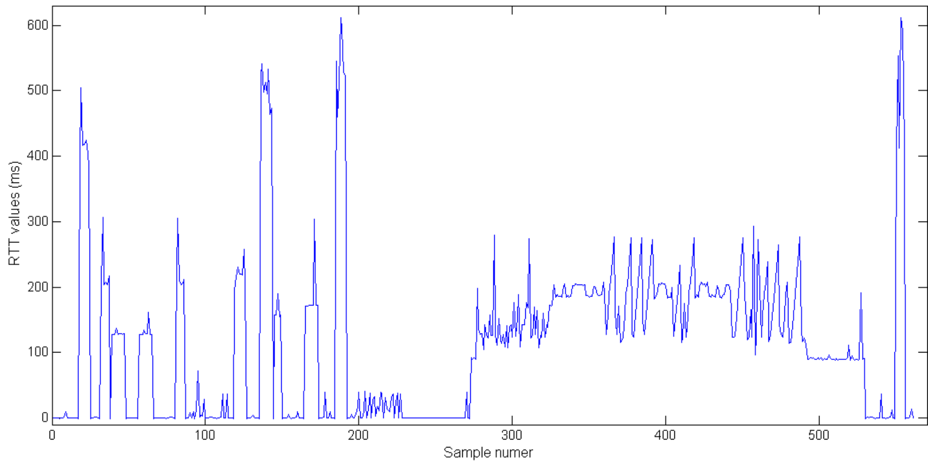


Fig. 3. Sample of RTT values for consecutive connections

An example of the sequence of RTT values measured in TCP connections is given in Figure 3 above. It is easy to realize that predicting the next value of RTT based on the past measurements, with a small error and few underestimates, is hard. A more complete characterization of real RTT traces can be found, for example, in [3].

3 Multi-objective Approaches for RTT Estimation

3.1 Multi-objective Optimization

We denote by Ω the space containing all candidate solutions to the given problem, RTT estimation in our case. In a *multi-objective optimization problem (MOP)* we want to find a candidate solution $\overrightarrow{x} \in \Omega$ which optimizes a vector of k *objective functions*: $\overrightarrow{f}(\overrightarrow{x}) = [f_1(\overrightarrow{x}), f_2(\overrightarrow{x}), \ldots, f_k(\overrightarrow{x})]$. The very same nature of a MOP implies that there may be many points in Ω representing practically acceptable solutions. It is often the case that the objective functions conflict with each other [8], e.g., a solution $\overrightarrow{x}$ could be better than another solution $\overrightarrow{y}$ for some of the k objectives while the reverse could be true for the remaining objectives.

An important definition for reasoning about solutions of a MOP is the *Pareto dominance* relationship:

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

$$\forall i \in \{1, 2, \ldots, k\}, \ f_i(\overrightarrow{u}) \leq f_i(\overrightarrow{v}) \ \textit{and} \ \exists i \in \{1, 2, \ldots, k\}, \ f_i(\overrightarrow{u}) < f_i(\overrightarrow{v}) \quad (2)$$

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

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

1. We define $ObjectiveFitness_1(\overrightarrow{u})$ as the average of the absolute distances between the sequence of $estimatedRTT$ constructed by solution $\overrightarrow{u}$ and the corresponding sequence of the $measuredRTT$ actually observed.
2. We define $ObjectiveFitness_2(\overrightarrow{u})$ as the number of times in which the *estimatedRTT* constructed by $\overrightarrow{u}$ is lower than the corresponding $measuredRTT$.

Both functions are evaluated on a set of training data collected as described in section 4.2. The ideal value for each of the two objective functions is zero. In the following subsections we describe the approaches that we have applied to this MOP.

In all the approaches we kept the nondominated solutions found during the evolutionary search. That is, at each generation we perform the following steps: (i) store in an external archive all the individuals nondominated by any other individual in the current population; (ii) drop from the archive individuals dominated by some other member of the archive.

3.2 Scalarization

This approach consists in combining the k objective functions in a single scalar objective F_{ws} to be minimized. The combination consists of a weighed sum of the objective functions with weights fixed a priori [7]:

$$F_{ws} = \sum_{i=1}^{k} w_i \cdot f_i \tag{3}$$

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

3.3 Pareto Dominance

We applied the Pareto dominance technique with a tournament selection scheme close to that in [9]. In the cited paper an individual is randomly picked from the population and then compared with a comparison set. Individuals that dominate the comparison set are selected for the reproduction. We modified this scheme according to the classical tournament selection, in which a group of n ($n \geq 2$) individuals is randomly picked from the population and the one with the best fitness is selected. Our scheme works as follows:

1. A tournament set of n ($n \geq 2$) individuals is randomly chosen from the population.
2. If one individual from the set is not dominated by any other individual, then it is selected.
3. Otherwise an individual is chosen randomly from the tournament set.

4 Experimental Procedure

4.1 RTT Traces

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

The resulting trace consists of 396109 RTT measures in 41521 TCP connections. These data are grouped in 78 files, for convenience. We chose one of these

files as *training set*, consisting of 5737 RTT measures on 611 TCP connections. The file selected as training set exhibits a large variety of scenarios: small and large variations, abrupt changes and so on. We used the remaining files, consisting of 390372 RTT measures on 40910 TCP connections, as *cross validation set*. The generalization capabilities of the solutions found on the training set have been evaluated on the cross validation set.

4.2 On Setting the GP Process

The terminal set and the function set is shown in Table 1. We used only arithmetic operators and a power of 2 as constant in order to obtain formulas that can be computed efficiently (this is a key requirement in TCP implementations and is necessary for fair comparison with the RTT estimator developed by Jacobson that is currently used). We allow the resulting formula to include the last measured RTT ($measuredRTT_{i-1}$) and the last estimation ($estimatedRTT_{i-1}$). We did not include values more far away in the past because the autocorrelation of RTT traffic is known to decrease very quickly [13].

Table 1. Terminals and functions set

Terminals set	$\frac{1}{2}, 1, measuredRTT_{i-1}, estimatedRTT_{i-1}$
Functions set	$+, -, /, \times$

Table 2. Parameter settings

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

For the first multi-objective approach we performed 25 independent executions for each combination of weights for a total of 225 runs, for the Pareto tournament scheme we performed the same amount of runs in order to carry out a fair comparison. Each execution starts with a different seed for the random number generator. We allocate 50 generations for each test. All others parameters are summarized in the Table 2. We used Sean Lukes Evolutionary Computation and Genetic Programming Research System (ECJ15) which is freely available on the web at `http://cs.gmu.edu/~eclab/projects/ecj/`. We modified and extended the original API for our needs.

5 Results

5.1 Comparison of the Multi-objective Approaches

In this section we compare the Pareto fronts generated by multi-objective genetic programming search based on the scalarization method and the Pareto-based tournament selection. A thorough comparison between the two approaches would require several indicators as those described in [18,6]. We use a much simpler comparison for lack of space and because both approaches exhibit significantly better performance than our baseline solution — the original algorithm by Jacobson.

We evaluated the performance on the cross validation dataset of the Jacobson algorithm and of each solution found with GP and belonging to the Pareto set. The results are summarized in Figure 4. The most important result is that

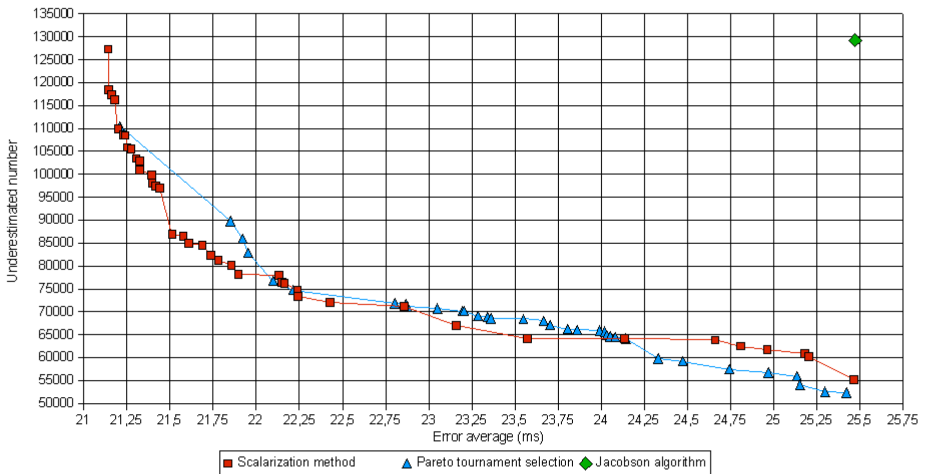


Fig. 4. Pareto front generated with the nondominated solutions for each multi-objective approach

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

5.2 Comparison of the RTT Estimators

To gain further insights into the quality of the solutions, in particular regarding the improvements that can be obtained with respect to the Jacobson algorithm,

we analyzed the performance on each single file of the cross validation dataset. We present the results for the Jacobson algorithm and for two solutions located at the two extremes of the Pareto front: the one giving the best results in terms of underestimated RTTs (located bottom-right in Figure 4) and the one giving the best results in terms of average error (located top-left).

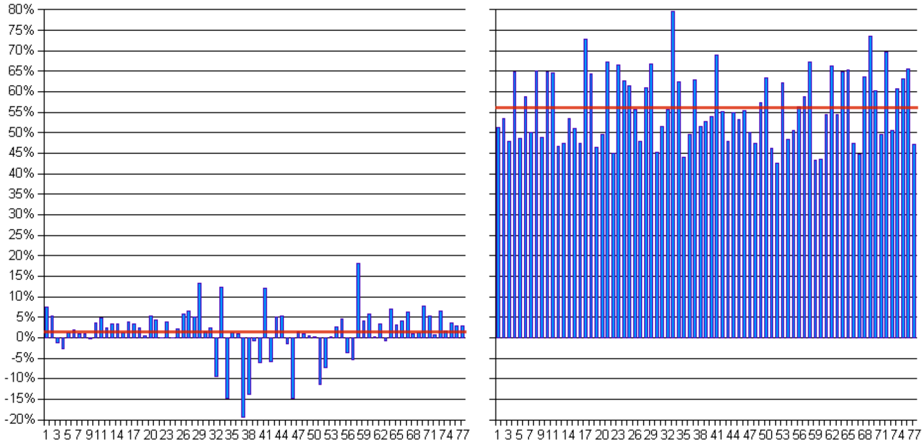


Fig. 5. Number of underestimated RTT for each trace file. Best formula found by GP in terms of average error (left) and in terms of number of underestimated RTTs (right)

Figure 5 describes the results in terms of underestimated RTTs. Each point in the X-axis corresponds to one file of the cross validation dataset, whereas the Y-axis is the improvement with respect to Jacobson, in percentage. The horizontal line represents the average improvement across the entire cross validation dataset. Figure 5-left shows the result for the best formula found by GP in terms of average error. It can be seen that the average improvement over the Jacobson algorithm is small (approximately 2%) and that in some files the average error is worse. Figure 5-right shows the result for the best formula found by GP in terms of number of underestimates. This case is much more interesting because the formula found by GP largely outperforms the Jacobson algorithm, with a 56% average improvement (34% of RTTs are underestimated by Jacobson and only 15% by the formula found by GP). Moreover, a remarkable improvement can be observed in every trace file.

Figure 6 describes the results in terms of average error, with the same notation as above. It can be seen the best formula in terms of average error (left figure) exhibit a 15% average improvement over Jacobson (corresponding to 4.7 ms) and that some improvement can be observed in every trace file. The best formula in terms of underestimated RTTs (right figure) exhibits instead essentially the same performance as that of Jacobson.

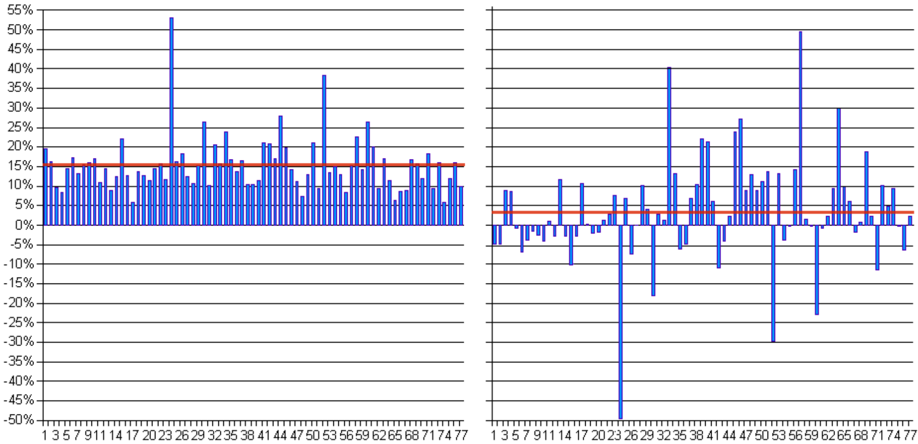


Fig. 6. Error average for each trace file. Best formula found by GP in terms of average error (left) and in terms of number of underestimated RTTs (right)

6 Concluding Remarks

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

While this is an interesting result itself, the most significant result consists in the performance of the formulas found with multi-objective GP: they are significantly better than those exhibited by the RTT estimator used in all TCP implementations. This result could lead to several interesting applications of GP in the networking field — e.g., tailoring RTT estimators to individual hosts, rather using the same estimator for all hosts; differentiating the estimator based on the application using TCP, whether web navigation or transmission of email; and so on.

Acknowledgments

The authors are grateful to Stefano Catani from the Centro Servizi Informatici di Ateneo (CSIA) for his technical help. This work is supported by the Marie-Curie RTD network AI4IA, EU contract MEST-CT-2004-514510 (December 14th 2004).

References

1. RFC 793, "Transmission Control Protocol,", Sept. 1981.
2. RFC 1122, "Requirements for Internet Hosts - Communication Layers," Oct. 1989.
3. J. Aikat, J. Kaur, F. D. Smith, K. Jeffay, "Variability in TCP round-trip times," in *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pp. 279–284, 2003.
4. M. Allman, V. Paxson, "On estimating end-to-end network path properties," in *ACM SIGCOMM Comput. Commun. Rev.*, Vol. 29, pp. 263-274, Oct. 1999.
5. W. Banzhaf, P. Nordin, R.E. Keller, F.D. Francone, "Genetic Programming - An Introduction; On the Automatic Evolution of Computer Programs and its Applications," Morgan Kaufmann, dpunkt.verlag, 1998.
6. P. A. N. Bosman, D. Thierens, The Balance Between Proximity and Diversity in Multiobjective Evolutionary Algorithms, in *Evolutionary Computation, IEEE Transactions*, Vol. 7, no. 2, pp. 174-88, Apr. 2003.
7. C. A. Coello Coello, "An Updated Survey of GA-based Multiobjective Optimization Techniques," in *ACM Computing Surveys*, Vol. 32, No. 2, Jun. 2000.
8. C. A. Coello Coello, D. A. Van Veldhuizen, G. B. Lamont, "Evolutionary Algorithms for Solving Multi-Objective Problems," Kluwer Academic Press, 2002.
9. A. Ekárt, S.N. Németh, "Selection Based on the Pareto Nondomination Criterion for controling Code Growth in Genetic Programming," in *Genetic Programming and Evolvable Machine*, Vol. 2, March 2001, pp. 61–73.
10. C. M. Fonseca, P. J. Fleming, "Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization," in *Proceedings 5th Int. Conf. Genetic Algorithms*, Ed. San Mateo, Morgan Kaufmann, pp. 416-423, 1993.
11. V. Jacobson, "Congestion avoidance and control," in *Proceedings ACM SIGCOMM*, Stanford, pp. 314-329, Aug. 1988.
12. J. R. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection," *MIT Press*, 1992.
13. L. Ma, K. E. Barner, G. R. Arce, "Statistical Analysis of TCP's Retransmission Timeout Algorithm," in *IEEE/ACM Transactions on Networking*, Vol. 14, Issue 2, Apr. 2006.
14. D. Parrott, X. Li, V. Ciesielski, "Multi-objective Techniques in Genetic Programming for Evolving Classifiers," in *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, Vol. 2, pp. 1141-1148, IEEE Press, 2-5 September 2005.
15. C. Poloni, "Hybrid Genetic Algorithm for Multiobjective Aerodynamic Optimisation," in *Genetic algorithms in engineering and computer science*, John Wiley & Sons, pp. 397–415, 1995.
16. N. Srinivas, K. Deb, "Multiobjective optimization using nondominated sorting in genetic algorithms," in *Evolutionary Computation*, Vol. 2, no. 3, pp. 221-248, 1994.
17. W. R. Stevens, G. R. Wright, "TCP/IP illustrated (vol. 2): the implementation", Addison-Wesley Longman Publishing Co., Inc., Boston, 1995.
18. E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, V. G. Fonseca, Performance Assessment of Multiobjective Optimizers: An Analysis and Review, in *Evolutionary Computation, IEEE Transactions*, Vol. 7, no. 2, pp. 117-132, Apr. 2003.

On Population Size and Neutrality: Facilitating the Evolution of Evolvability

Richard M. Downing

University of Birmingham
rmd@cs.bham.ac.uk

Abstract. The role of population size is investigated within a neutrality induced local optima free search space. Neutrality decouples genotypic variation in evolvability from fitness variation. Population diversity and neutrality work in conjunction to facilitate evolvability exploration whilst restraining its loss to drift, ultimately facilitating the evolution of evolvability. The characterising dynamics and implications are discussed.

1 Introduction

Evolutionary search is traditionally depicted as taking place on a multi-modal landscape: If population diversity can be maintained, then the search will progress towards optimal fitness along the gradients of many optima simultaneously, leaving local optima in its wake. Accompanying this depiction are the difficulties of diversity maintenance and sensitivity to the starting conditions, both of which remain prominent issues within the EC research community.

However, neutrality [13] proponents argue a different perspective on evolutionary search. In the neutralist depiction, neutral networks alleviate local optima and the loss of diversity is of little concern. For example, Ebner et al. [9] find that redundant representations increase accessibility between phenotypes through neutral walk. Harvey & Thompson [10] show that evolution can progress satisfactorily in a small, genetically converged population for an evolutionary hardware task. Barnett [2] goes further still and argues that a non-population based approach is optimal. Studies employing RNA models have been particularly influential [12,17,11], the structure of RNA spaces apparently exhibiting the purported properties more readily than artificial representations. The potential of neutrality has been further recognised in [20,21], though others voice more sceptical or cautionary notes [16,15,19].

So, what is the role of population diversity in a neutrality-induced local optima free search space? While the search space is proven to be free of local optima for the EA and problems employed for this investigation [6], it is clearly not uni-modal in the intuitive sense of a hill with a single peak. The space is highly neutral and perforated by massively connected neutral networks. Should population diversity be considered beneficial in such a space, or should a non-population based approach be preferred as it would be in the intuitive idea of a single-peaked space? These are the questions addressed by this study.

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 181–192, 2007.
© Springer-Verlag Berlin Heidelberg 2007

The conclusions identify neutrality and population diversity working in conjunction to facilitate the evolution of evolvability in a three step process:

1. Neutral mutations create evolvability discrepancies whilst conserving fitness.
2. Population size facilitates evolvability exploration, but also restrains drift away from favourable evolvability characteristics.
3. Selection acts indirectly, through fitness, on evolvability, propagating the more favourable evolvability characteristics.

This depiction promotes evolvability as the principal beneficiary of genotypic variation and selection. Whilst selection acts directly on fitness, it is evolvability that is the ultimate target [5, Appendix]. Effectively, fitness becomes evolvability's selection surrogate, the expression of evolvability's latent potential, exposing to selection that which evolvability, by itself, cannot. In this, evolvability can emerge, and the evolution of evolvability be witnessed.

In contrast, Barnett [2] found that a variety of hill-climber, not a population-based approach, proved optimal; and Smith et al. [19] found that evolvability was not evolving during neutral evolution. The reason for these contradictory conclusions is not examined here, but differences in the representations and operators are likely to be primarily responsible. What makes the representation employed in this paper so different in its search characteristics is the subject of ongoing investigation.

In this paper, *evolvability* is considered a primarily genotypic property, and is defined as "the heritable *potential* to acquire increased fitness through random mutation". Evolvability as a property of the population as a whole, epitomised by Altenberg's [1] definition, is also relevant in this paper, though secondary.

Sections 2 & 3 review the representation and EA employed. Section 4 elucidates the potential of the evolution of evolvability. Sections 5 & 6 examine the effects of losing diversity, concluding the employment of greedy selection. Sections 7 & 8 investigate the potential of diversity and neutrality on functions contrasting in their evolvability potential.

2 Binary Decision Diagrams

An *ordered* BDD (OBDD) is a rooted directed acyclic graph representing a function of the form $f(V) : \mathbb{B}^n \longrightarrow \mathbb{B}$. Each non-terminal is labeled with a Boolean variable $v \in V$ and has a *then* child and an *else* child, reflecting the fact that each non-terminal represents an *if-then-else* operation on v. Terminals are labeled from $\mathbb{B}$. A total ordering is imposed on the appearance of non-terminal labels along all paths with π, the variable ordering. Thus, $\pi = [v_1, v_2, \ldots, v_n]$, an ordered list of variables, and $i < j$ must hold for each v_i followed by v_j along any path. It is not necessary that all $v \in \pi$ appear in a path.

Redundancy in an OBDD can be removed by removing any non-terminals that have both child edges pointing to the same vertex, or by merging two vertices that have both the same label and the same respective children. A *reduced* OBDD (ROBDD) is an OBDD that cannot have its complexity reduced further by these

reductions. Bryant [4] has shown ROBDDs to be *canonical forms*; meaning that each function has a unique ROBDD representation for a given π, reachable by the above reductions.

The variable ordering, π, can have a dramatic impact on the *complexity* of resulting ROBDD: In this paper, the complexity of an π-ROBDD is the number of nonterminals it contains. For example, the best π for the 11-bit multiplexer produces an ROBDD having complexity 15; the worst, 509: For the 20-bit multiplexer the best π produces an ROBDD having complexity 31; the worst, over 130,000. For the n-bit multiplexer, the complexity grows linearly for the best π and exponentially for the worst. Generally, the variable ordering problem is known to be NP-complete in both optimal and approximate solutions [3,18]. Refer to Bryant [4] for further details on BDDs.

3 Algorithm

The EA for evolving BDDs is now reviewed and its relevant characteristics discussed: From hereon it will be referred to by the acronym EBDDIN after the title of the paper that introduced it [6]. A recent enhancement to EBDDIN exhibits the emergence of good variable orderings for the multiplexer problem [7], and this property will be exploited in later sections.

EBDDIN is built on the following mutations on genotypes in OBDD space, five of which are explicitly neutral and one of which is potentially adaptive. The neutral mutations are derived from established OBDD theory; the functionally modifying mutation is a natural and intuitive one for graph-based representations. Further details can be found in [6,7,8].

Definition 1. *Let* **N1** *be the neutral mutation of removing a redundant test.*

Definition 2. *Let* **N1′** *be the neutral mutation of inserting a redundant test, the inverse of* **N1**.

Definition 3. *Let* **N2** *be the neutral mutation of removing a redundant nonterminal, merging two (or more) nonterminals.*

Definition 4. *Let* **N2′** *be the neutral mutation of creating a redundant nonterminal, splitting a non-terminal.*

Definition 5. *Let* **N3** *be the neutral mutation of swapping adjacent variables while maintaining overall function.*

Definition 6. *Let* **A1** *be the 'potentially' adaptive mutation of changing one of the children of a non-terminal, to another vertex.*

Firstly, note that, because neutral mutations are defined explicitly, the function evaluation can be circumvented where a child is generated from a neutral mutation. Only one mutation is ever used to produce an offspring [8], so many offspring will be neutral, the number of which is dependent on the relative frequency of applying the different mutations. Effectively, this results in an *effortless neutral walk*

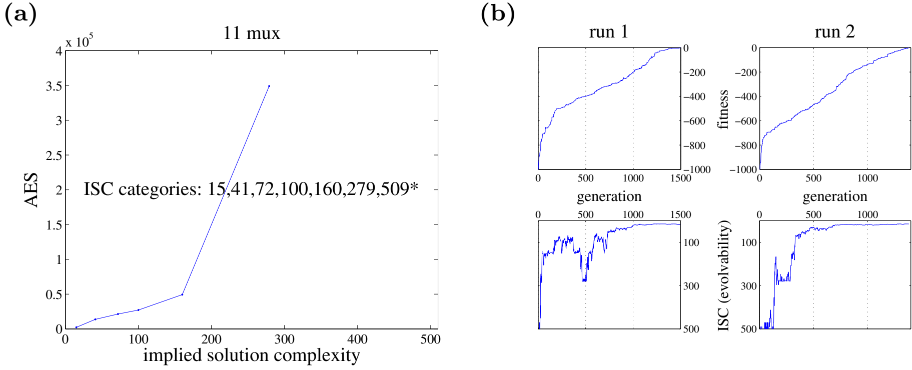


Fig. 1. (a) Comparison of fixed ISC categories for a fixed ordering (i.e. no N3 mutation). π are categorised by ISC value. Rapidly increasing AES against increasing ISC is observed. *No AES value could be obtained for ISC = 509. (b) Fitness curves resulting from the evolution of evolvability. An almost constant rate of fitness increase is maintained for the duration of the run as evolvability emerges in the form of low ISC.

where effort is determined by the number of evaluations required to generate the target function.

Secondly, performance is highly dependant on the nature of the target function with regard to its ROBDD complexity. The relationship between evolvability and π was investigated in [7] for a number of functions. The effort required to solve a problem, in terms of the *Average number of Evaluations to a Solution* (AES), grew super-linearly in the *implied solution complexity* (ISC) of π for a fixed ordering: ISC is the complexity of ROBDD induced by a given π for a given target function [7][1]. For example, the parity function always evolves rapidly because ISC is constant and linear in the number of variables for all π; the multiplier function always evolves poorly because ISC is exponential in the number of variables for all π. See figure 1(a) for how evolvability on the multiplexer function changes with varying ISC. When dynamic variable reordering is introduced through the N3 mutation good π will emerge for the multiplexer [7]. Thus, evolvability is dependant on π, and good π with low ISC values can emerge where discrepancies in ISC exist for a function. The contrasting properties of the parity and multiplexer functions in this respect will be exploited in the experiments discussed in the following sections.

Finally, the search space for this representation, under these mutations, for a minimal mutation rate, and for the type of target functions employed, is known to be free of local optima where local optima is defined in terms of accessability [6]. It should also be noted that each function is associated with a single, fully-connected neutral network of genotypes, none of which are functionally isolated:

[1] ISC is calculated using a representation of the target function. The target function has its variables reordered to match a given π or the ordering of a candidate solution. The number of non-terminals in the ROBDD representing the target then provides the ISC for that variable ordering.

That is, for every function in OBDD space, all the genotypes representative of a given function are connected via a neutral walk, but each genotype also has mutational neighbours (through A1) which differ in function, making the neutral networks highly intertwined.

4 The Evolution of Evolvability: The Promise

The promise offered by the evolution of evolvability is now elucidated by the presentation of some linear fitness curves using the 11-bit multiplexer as the target function. The curves are plotted alongside ISC to provide an indication of how evolvability and fitness correspond. A $(7, 12)$ ES is employed and the population initialised with individuals having worst possible ISC. See figure 1(b).

What is impressive about these fitness curves is that they maintain an almost linear fitness increase for a prolonged period, and do so in a gradualistic manner. It is the evolution of evolvability that facilitates this behaviour. The simultaneous increase in evolvability indicated by the emerging lower ISC values maintains the rate of fitness improvement against a search space becoming increasingly sparse in superior solutions. The fact that such curves can be induced at all is something not readily seen in EC, and it is indicative of sustainable and progressive evolution.

5 Constraining Diversity

The constraining of diversity and its influence on performance is now investigated. Three experiments are conducted on both the 10 parity and 11-bit multiplexer functions. A population of 30 is first generated and written to disk. The results are plotted in figure 2(a).

For the first experiment, the population is read from disk and the number of evaluations required to solve the problem plotted over 30 runs. For the second experiment, the population is initialised to clones for each of the 30 runs, one run for each of the individuals on disk. For the final experiment, the setup is similar to that described for the second experiment but, additionally, diversity is periodically removed every 50 generations by only breeding one parent for that generation. A $(15 + 30)$ ES is employed. The setup facilitates comparison of a population that is not prevented from maintaining initial diversity, a population that has no initial diversity, and a population that has diversity periodically eliminated.

For both problems and all configurations a 100% success rate is maintained. Furthermore, the effect on the number of evaluations required from the loss of diversity is negligible; it is slightly accentuated for mux where the population is initialised to be diverse, but this can be attributed to the higher probability of having better evolvability (i.e. lower ISC values) present in at initialisation rather than having to wait for it to emerge. These results suggests a certain uniformity in the search space and the search can be considered highly independent of the

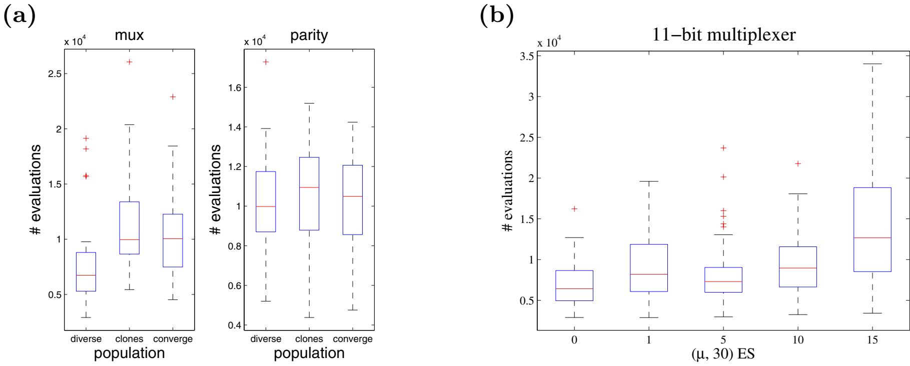


Fig. 2. (a) Comparison of performance with various restrictions on diversity. *diverse* - initial population of unique individuals; *clones* - initial population of clones; *converge* - periodically remove diversity every 50 generations. (b) How greedy selection compares against standard truncation selection.

starting configuration. Furthermore, the genotypic convergence accompanying a fitness improvement step can be considered benign.

6 Fitness Conservation and Generation Lag

Given that temporary loss of diversity has negligible effect on performance (section 5), maximising selection pressure can be considered. Altenberg [1] has emphasised the importance of strong parent to offspring fitness correlation for evolvability; achieving this through neutral mutation and selecting only the fittest individuals to be parents has great appeal. This type of *greedy selection* will be denoted $(0, \lambda)$ or $(0 + \lambda)$ in the style of ES for generational algorithms. The $\mu = 0$ indicates that the number of parents is not specified explicitly but depends on the the number of individuals currently exhibiting the equal highest fitness, which may vary between 1 and λ.

Figure 2(b) examines how a $(0, 30)$ ES compares against standard selection, and the former is found to be favourable. Thus, the benefits of not breeding suboptimal solutions outweighs any loss in genotypic diversity from fitness diversity, emphasising the potential of neutrality to decouple genotypic variation from fitness variation, the significance of which is well-recognised [20,11,17].

The effects of *generation lag* must also be recognised when using AES as the performance measure. Generation lag occurs at the fitness improvement step during the generation of the child population. A fitness improvement early in the production of the child population is not available for breeding until the following generation, resulting in the breeding of suboptimal solutions until the child population is fully populated. The cost increases with population size and must be balanced against any beneficial effects of a larger population. In the following experiments generation lag will be recognised as a consequence of a generational algorithm, or eliminated with a steady-state variant where indicated.

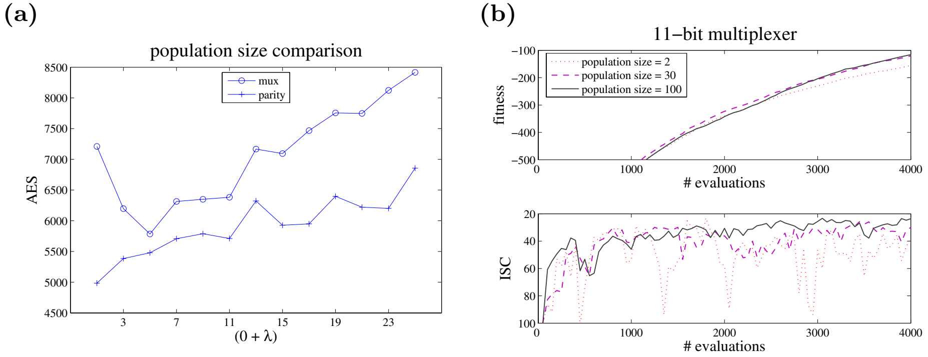


Fig. 3. (a) Effects of population size on performance (100 runs). Where there is potential for evolvability variation (mux), population diversity can improve performance, but generation lag taints the results. (b) Trace on population sizes averaged over 10 runs using greedy steady-state selection (no generation lag). A larger population is better able to maintain low ISC.

7 Evolvability Discrepancies

The effect of evolvability discrepancies is now investigated, and an *evolvability error threshold* postulated. The objective is to identify population diversity as a promoter of favourable evolvability characteristics rather than maintaining gross genotypic diversity from initialisation. Whilst the former can easily arise gradually from a population of clones through only a minor mutation severity, the latter cannot.

Recall that low ISC indicates greater evolvability and the multiplexer function exhibits significant discrepancies in evolvability as a result of π (figure 1a). The parity function, however, exhibits no such discrepancies because it is a symmetric function, exhibiting constant ISC for all π. Figure 3(a) shows the difference in performance characteristics that result from the presence of evolvability discrepancies for different population sizes. From the figure, it is clear that the parity function exhibits an increasing AES against population size. The increase appears roughly linear, and can be attributed to generation lag. For mux, a performance gain is observed up until a population size of around 5, at which point performance takes a downturn. Clearly, where there are evolvability discrepancies to be attained, a larger population appears to offer some benefit, but the presence of generation lag taints the results. Thus, in the following experiments, steady-state selection is employed.

Figure 3(b) shows a trace averaged over 10 runs using greedy steady-state selection. The figure shows that a larger population is better able to maintain evolvability (low ISC values), and this corresponds to more rapid fitness improvement. The distribution of ISC for mux is heavily skewed, with an expectation of approximately 50, so the consequence on fitness of losing ISC is not great

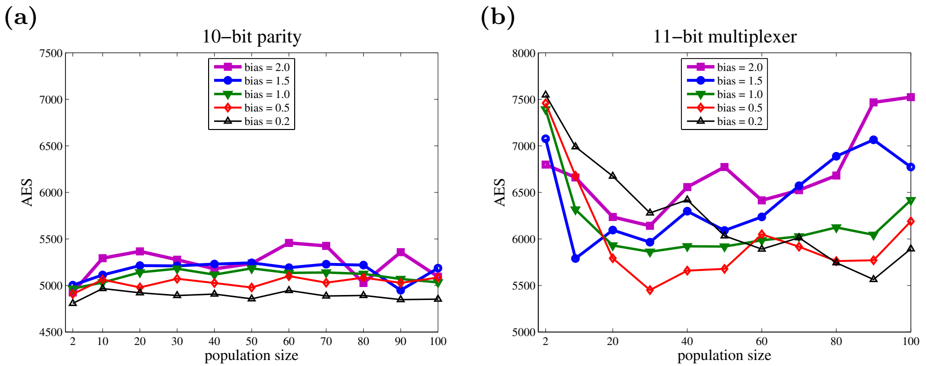


Fig. 4. The effect on performance of population size for varying degrees of neutrality. (a) Parity - increasing population size has negligible effect on performance due to the absence of discrepancies in evolvability. More neutrality improves performance; (b) Multiplexer - increasing population size has a sometimes positive effect on performance, particularly where a higher degree of neutrality is present. See text for a full discussion.

(refer to figure 1). A function having a differing ISC distribution, however, would have adaptation much more grossly impaired.

Thus, an *evolvability error threshold* can be postulated, at which point evolvability is lost to drift. Huynen et al. [12] discuss genotypic versus phenotypic error thresholds, stating that it is the latter at which adaptation breaks down. However, these results suggest adaptation can also be stifled at the evolvability threshold whilst maintaining the phenotype.

8 Population Size and Neutrality

The relationship between diversity and neutrality is now investigated. Figure 4 shows the effect of varying both population size and the degree of neutrality for both the parity and mux functions. Again, a greedy steady-state selection is employed. The degree of neutrality is influenced by the ratio of adaptive to neutral mutations which, in turn, is controlled by introducing a *mutation bias*: A bias of 2.0 means that it is twice as likely that the adaptive A1 mutation is chosen for application than it would be otherwise; a bias of 0.5 means that A1 is half as likely to be chosen. Thus, a lower bias means a lower ratio of adaptive to neutral mutations and, therefore, more neutral offspring. In the figures, the size of the bias is reflected by heavier lines, so lighter lines reflect the fact that comparatively more neutral drift is present. The population is initialised to clones to eliminate any diversity oriented initialisation benefit for larger populations; refer to figure 2(a) for an indication of how this can affect performance.

For the parity function, figure 4(a), increasing population size has negligible effect on performance and variance in AES is around 5%. This behaviour is expected given that there are known to be no discrepancies in evolvability resulting from exploration of π; one individual is just as evolvable as another. However,

an increase in neutral drift through a lower mutation bias improves performance independently of population size: More neutral drift means greater exploration generally and less effort re-sampling familiar areas of the space.

In contrast, for the multiplexer function, figure 4(b), trends against population size are discernable and variance in AES is much greater. Increasing population size results in a decrease in AES followed by an increase in AES in most cases: Too large a population stifles neutral drift and, therefore, exploration. At the smallest population size, more neutrality consistently results in poorer performance, favourable π being more readily lost to neutral drift. At the largest population, however, the situation is completely reversed; it is able to tolerate greater neutrality-induced exploration while maintaining evolvability, resulting in overall better performance where neutrality is higher. This result suggests a certain synergy in increasing both neutrality and population size with the graph exhibiting most neutrality approaching monotonic.

9 Discussion

It should be noted that these results are particularly dependent on the AES performance measure and EBDDIN's capability for effortless neutral walk; the cost of evaluating neutral offspring would have to be balanced against the benefits otherwise. This advantage makes EBDDIN more akin to natural systems than less so. In nature, the evaluation of a population's fitness is highly parallel, and a greater propensity for neutral offspring can be reflected in an expanding and subdividing population. In contrast, evaluation of a population in artificial evolution is typically done serially, as reflected by the AES performance measure, and the population size and number is also typically fixed. Thus, there is a *serial evaluation deficiency* associated with artificial evolution. EBDDIN's capability to circumvent the evaluation cost of many offspring alleviates some of this deficiency.

Repeating these experiments with other representations may be difficult, but this does not detract from the generality of the results. For EBDDIN, a significant degree of evolvability has been identified as being associated with the quality of the variable ordering. This allows evolvability to be monitored during a run via ISC, and varied randomly along that dimension via neutral mutation. For other representations identifying such a feature of the genotype that is associated with evolvability is not so easy, perhaps. However, that does not mean that evolvability cannot emerge in a similar way; it means only that evolvability cannot be targeted and observed so readily. Indeed, contemporary evolutionary thinking suggests that evolvability did emerge: For example, Kirschner & Gerhart's [14] theory of "facilitated variation" claims that *core processes* emerged early in the origin of life, and that these constrain variation to maintain high offspring viability and the variational aspect of evolvability. It is reasonable to assume that evolvability can emerge in artificial representations too, irrespective of whether or not particular genotypic features that are important to, or indicative of, evolvability can be readily identified.

The generality of these results may further be questioned because they are conducted in a local optima free search space. However, regardless of the modality in the search space, evolvability discrepancies within a population are clearly significant and a prerequisite for the evolution of evolvability, a property considered fundamental in the evolution of complex systems. Indeed, who is to say that, for a lineage to progress satisfactorily towards ever-greater complexity, local optima sparsity, neutrality-induced or otherwise, is not a prerequisite? Perhaps, rather than pursuing efforts to maintain diversity in highly-modal and deceptive spaces, the EC community might better focus on transforming the search space of a problem, with the aid of neutrality, into one more susceptible to gradual evolution? This work provides insights into the role of population diversity in such spaces and will aid those researchers adopting such an approach.

10 Summary

Where the target function exhibits attainable discrepancies in evolvability (i.e. mux), a larger population is better able to exploit them. Where there are no such discrepancies (i.e. parity), a larger population offers no benefit, but no detriment either when steady-state selection is employed. Population size serves the dual role of facilitating evolvability exploration and restraining its loss to drift. The effectiveness with which these two roles are satisfied depends on the degree of neutrality present.

Current and future research focuses on enhancing EBDDIN by: evolving the relative frequency of applying the different mutations for even greater attainments in evolvability; and employing a wider variety of variable reordering operators to tackle a more diverse range of problems. The question of what makes one genotype more evolvable than another is also under investigation.

Acknowledgements

Thank you to Ata Kaban for raising some of the questions that motivated this paper, and to the anonymous reviewers for their helpful comments.

References

1. Lee Altenberg. The evolution of evolvability in genetic programming. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 3, pages 47–74. MIT Press, 1994.
2. Lionel Barnett. Netcrawling - optimal evolutionary search with neutral networks. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 30–37, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.
3. B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

4. Randall E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
5. Richard Dawkins. *The Blind Watchmaker.* Penguin Books, London, 1991.
6. Richard M. Downing. Evolving Binary Decision Diagrams using implicit neutrality. In David Corne, Zbigniew Michalewicz, Marco Dorigo, Gusz Eiben, David Fogel, Carlos Fonseca, Garrison Greenwood, Tan Kay Chen, Guenther Raidl, Ali Zalzala, Simon Lucas, Ben Paechter, Jennifier Willies, Juan J. Merelo Guervos, Eugene Eberbach, Bob McKay, Alastair Channon, Ashutosh Tiwari, L. Gwenn Volkert, Dan Ashlock, and Marc Schoenauer, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 3, pages 2107–2113, Edinburgh, UK, 2-5 September 2005. IEEE Press.
7. Richard M. Downing. Evolving binary decision diagrams with emergent variable orderings. In Thomas Philip Runarsson, Hans-Georg Beyer, Edmund Burke, Juan J. Merelo-Guervos, L. Darrell Whitley, and Xin Yao, editors, *Parallel Problem Solving from Nature - PPSN IX*, volume 4193 of *LNCS*, pages 798–807, Reykjavik, Iceland, 9-13 September 2006. Springer-Verlag.
8. Richard M. Downing. Neutrality and gradualism: encouraging exploration and exploitation simultaneously with binary decision diagrams. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 615–622, Vancouver, 6-21 July 2006. IEEE Press.
9. Marc Ebner, Patrick Langguth, Juergen Albert, Mark Shackleton, and Rob Shipman. On neutral networks and evolvability. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 1–8, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.
10. I. Harvey and A. Thompson. Through the labyrinth evolution finds a way: A silicon ridge. In T. Higuchi, M. Iwata, and L. Weixin, editors, *Proc. 1st Int. Conf. on Evolvable Systems (ICES'96)*, volume 1259 of *LNCS*, pages 406–422. Springer-Verlag, 1997.
11. Martijn A. Huyen. Exploring phenotype space through neutral evolution. *Journal of Molecular Evolution*, 43(3):165–169, September 1996.
12. Martijn A. Huynen, Peter F. Stadler, and Walter Fontana. Smoothness within ruggedness: the role of neutrality in adaptation. *Proc. Natl. Acad. Sci. (USA)*, 93:397–401, 1996. SFI preprint 95-01-006, LAUR-94-3763.
13. Motoo Kimura. *The neutral theory of molecular evolution.* Cambridge University Press, 1983.
14. Marc W. Kirschner and John C. Gerhart. *The Plausibility of Life: Resolving Darwin's Dilemma.* Yale, 2005.
15. Joshua D. Knowles and Richard A. Watson. On the utility of redundant encodings in mutation-based evolutionary search. In H.-G. Beyer J.-L. Fernández-Villacañas H.-P. Schwefel J.-J. Merelo Guervós, P. Adamidis, editor, *Parallel Problem Solving from Nature - PPSN VII, 7th International Conference, Granada, Spain, September 7-11, 2002. Proceedings*, number 2439 in Lecture Notes in Computer Science, LNCS, page 88 ff. Springer-Verlag, 2002.
16. Franz Rothlauf and David E. Goldberg. Redundant representations in evolutionary computation. *Evol. Comput.*, 11(4):381–415, 2003.
17. Peter Schuster, Walter Fontana, Peter F Stadler, and Ivo L Hofacker. From sequences to shapes and back: A case study in RNA secondary structures. *Proc. Roy. Soc. Lond. B*, 255:279–284, 1994.
18. D. Sieling. On the existence of polynomial time approximation schemes for OBDD-Minimization. *LNCS*, 1373:205–215, 1998.

19. Tom Smith, Phil Husbands, and Michael O'Shea. Neutral networks and evolvability with complex genotype-phenotype mapping. In J. Kelemen and P. Sosík, editors, *ECAL 2001*, volume 2159 of *Lecture Notes in Computer Science*, pages 272–281, Berlin Heidelberg, 2001. Springer-Verlag.
20. Marc Toussaint and Christian Igel. Neutrality: A necessity for self-adaptation. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2002)*, pages 1354–1359, 2002.
21. Tina Yu and Julian Miller. Neutrality and the evolvability of boolean function landscape. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 204–217, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

On the Limiting Distribution of Program Sizes in Tree-Based Genetic Programming

Riccardo Poli, William B. Langdon, and Stephen Dignum

Department of Computer Science, University of Essex, UK

Abstract. We provide strong theoretical and experimental evidence that standard sub-tree crossover with uniform selection of crossover points pushes a population of a-ary GP trees towards a distribution of tree sizes of the form:

$$\Pr\{n\} = (1 - ap_a)\binom{an+1}{n}(1 - p_a)^{(a-1)n+1}\, p_a^n$$

where n is the number of internal nodes in a tree and p_a is a constant. This result generalises the result previously reported for the case $a = 1$.

1 Introduction

For most problems the ratio between the size of the search spaces and the number of acceptable solutions grows exponentially with the size of the problem. So, even with today's powerful computers, for many problems one can hope to find solutions with a particular search algorithm only if the algorithms is biased in such a way to sample preferentially the areas of the search space where solutions are denser. This situation is often informally referred to as an *algorithm being well-matched to a problem*.

Having a full characterisation of the search biases of a search algorithm is a precondition to understand whether or not the algorithm is well-matched to a problem. (The second precondition is the availability of a characterisation of the problem, e.g., information on the distribution of solutions in the search space.)

In evolutionary algorithms this requires understanding the biases of the genetic operators. These biases are fairly well understood for mutation and crossover in the case of fixed-length representations (e.g., binary GAs) [4,16] and for selection (which is representation independent) [5,1,2,11]. However, the situation is much sketchier for variable-length representations. In particular, except for the limiting case of linear-trees (built only using arity-1 primitives and terminals) [12,15,13,14], we still know very little about the search biases of standard GP crossover.

In this paper we provide an exact characterisation of the limiting distribution of tree sizes towards which sub-tree crossover, when acting on its own, pushes the population. As we will see, obtaining this type of result is complex, and so we will limit our attention to the case where the primitive set includes only terminals and primitives of one other arity.

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 193–204, 2007.
© Springer-Verlag Berlin Heidelberg 2007

The paper is organised as follows. In Section 2 we summarise results from branching processes theory. In Section 3 we develop a formulation for the fixed-point distribution of tree sizes under repeated crossover. In Section 4 we derive an equation that describes how the distribution of tree sizes changes generation after generation under the effects of crossover. The distribution proposed in Section 3 is one of the two elements of such an equation, when evaluated at its fixed-point. In Section 5 we develop an explicit formulation for the second element: the distribution of subtree sizes. In principle these ingredients would allow one to check mathematically whether the proposed size distribution is indeed a fixed point for crossover-based evolution. Proving this result is, however, beyond our mathematical capabilities. Therefore, to corroborate our conjecture we present strong empirical evidence and numerical integrations of the tree-size distribution evolution equation in Section 6. We make some final remarks in Section 7.

2 Mathematical Preliminaries

2.1 Branching Processes and Lagrange Distribution

In probability theory, a *discrete-time branching process* [17] is a Markov process that models a population in which each individual in a generation produces some random number of descendants, and where the probability of generating a successors, $p(a)$, is fixed. This leads to a (family) tree.

Branching processes have at least one application in GP: if no limit is imposed on tree size or depth, the tree shapes produced by the "grow" method, often used to initialise GP populations and to perform sub-tree mutation, obey a branching process. In this case a is the arity of primitives and p_a is the probability of using primitives of arity a when choosing nodes in the "grow" method.

The distribution of tree sizes for a branching process follows a *Lagrange distribution* [3,6]. More precisely, the probability of the process leading to a total of ℓ individuals being generated is

$$\Pr\{L=\ell\} = \begin{cases} 0 & \text{if } \ell = 0, \\ \frac{1}{\ell}\mathcal{C}(t^{\ell-1})\left\{(g(t))^{\ell}\right\} & \text{for } \ell = 1,2,3,\cdots, \end{cases} \tag{1}$$

where $g(t) = \sum_a p_a\, t^a$ is the probability generating function of the distribution p_a and $\mathcal{C}(t^m)$ denotes "the coefficient of t^m in".

If one considers a process where only nodes of arity a and 0 are allowed (i.e., $p_0 + p_a = 1$), then $g(t) = p_0 + p_a t^a$. So, for $\ell > 0$ we have

$$\mathcal{C}(t^{\ell-1})\left\{(g(t))^{\ell}\right\} = \mathcal{C}(t^{\ell-1})\left\{(p_0 + p_a t^a)^{\ell}\right\} = \mathcal{C}(t^{\ell-1})\left\{\sum_{k=0}^{\ell}\binom{\ell}{k} p_0^{\ell-k}\, p_a^k\, t^{ak}\right\}$$

Since $\mathcal{C}(t^{\ell-1})$ will pick out the coefficient of the power of t for which $\ell-1 = ak$, i.e., $k = \frac{\ell-1}{a}$, we then have

$$\Pr\{L=\ell\} = \begin{cases} \frac{1}{\ell}\binom{\ell}{\frac{\ell-1}{a}}(1-p_a)^{\ell-\frac{\ell-1}{a}}\, p_a^{\frac{\ell-1}{a}} & \text{if } \ell-1 \text{ is a multiple of } a, \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Note that, since only arity 0 and arity a primitives are allowed, a tree with $\ell - 1$ nodes has $n = \frac{\ell-1}{a}$ internal nodes and $\ell = an + 1$. So, we can rewrite the previous equation in terms of internal nodes as

$$\Pr\{N = n\} = C_{\mathcal{T}}(a, n)\ (1 - p_a)^{(a-1)n+1}\, p_a^n, \tag{3}$$

where

$$C_{\mathcal{T}}(a, n) = \frac{1}{an + 1}\binom{an + 1}{n} \tag{4}$$

is a *generalised Catalan number* [7]. The Catalan number is the number of different trees with n internal nodes of arity a (and, of course, $(a-1)n+1$ leaves). This can be interpreted as saying that in a branching process all trees of a particular size have a probability of being created which depends only on how many nodes/primitives of each kind the tree contains. It also means that the different parts of the trees created by a branching process are uncorrelated.

2.2 Moments of the Tree-Size Distribution in a Branching Process

It is possible to compute the moments of Lagrange distributions starting from the the cumulants g_i of the probability density functions generated by power series in t of $g(t)$ [6,3]. The mean progeny produced by a branching process is:

$$E[L] = \frac{1}{1 - g_1} \tag{5}$$

and the variance is

$$Var[L] = \frac{g_2}{(1 - g_1)^3}, \tag{6}$$

where $g_1 = E[A]$ and $g_2 = Var[A]$, A being a stochastic variable representing a node's arity. Since trees contain only arity 0 and arity a nodes, we can easily compute these two cumulants:

$$g_1 = \sum_k k p_k = a p_a \tag{7}$$

$$g_2 = E[A^2] - (E[A])^2 = a^2 p_a - (a p_a)^2 = a^2 p_a (1 - p_a) \tag{8}$$

So, the mean tree size in our branching process is

$$E[L] = \frac{1}{1 - a p_a} \tag{9}$$

and the variance is

$$Var[L] = \frac{a^2 p_a (1 - p_a)}{(1 - a p_a)^3}. \tag{10}$$

From these two, we then obtain the second non-central moment

$$E[L^2] = Var[L] + (E[L])^2 = \frac{(a-1)a p_a - a^2 p_a^2 + 1}{(1 - a p_a)^3} \tag{11}$$

Note that (9) matches the formula for the means size of programs built by the "grow" method reported in [10] and that (9) is a special case of it.

3 The Distribution of Tree Sizes Under Crossover

In the absence of selection, if a population of GP trees undergoes repeated crossovers, the population tends to a limiting distribution of sizes and shapes. This is the result of the specific bias of subtree crossover.[1] Effectively after a while, every node in every individual in the population will have been placed at its particular position as a result of one or multiple crossover events. So, any correlations present in the shapes in the initial generation will have been broken by crossover.

As we saw in the previous section, complete decorrelation in the different parts of a tree is a characteristic of branching processes. Within the class of trees of a given size, each shape is equally likely. So, we postulate that the limiting distribution of tree sizes under repeated crossover will be one where this happens. That is, we assume that at the fixed-point, the shape distribution is

$$\Pr\{\text{Shape with } n \text{ nodes of arity } a\} = w(n,a)\,(1-p_a)^{(a-1)n+1}\,p_a^n \tag{12}$$

where $w(n,a)$ is an appropriate sequence of weights to be determined and p_a is a parameter, also to be determined. So, the probability of picking a tree with n internal nodes from the population is

$$\Pr\{n\} = C_{\mathcal{T}}(a,n)\,w(n,a)\,(1-p_a)^{(a-1)n+1}\,p_a^n \tag{13}$$

What constraints do we have on the parameters $w(n,a)$ and p_a? Firstly, they must be such that the distribution of shapes is indeed a probability distribution. In particular we require

$$\sum_{n\geq 0} \Pr\{n\} = 1. \tag{14}$$

Secondly, it is well-known that on average subtree crossover does not alter the mean size of program trees in a population [14]. So, we also require that

$$\sum_{n\geq 0} (an+1)\Pr\{n\} = \mu_0, \tag{15}$$

where μ_0 is the average size of the individuals in the population at generation 0. Thirdly, we require (13) to be a generalisation of the results reported in [12,15,13,14] for arity 1 functions, which we here rewrite as

$$\Pr\{\ell\} = \ell r^{\ell-1}(1-r)^2, \tag{16}$$

where

$$r = (\mu_0 - 1)/(\mu_0 + 1). \tag{17}$$

We can do this by setting $a = 1$ in (13) and $\ell = n+1$ and so $\Pr\{\ell\}$ in (16) is the same quantity as $\Pr\{n\}$ in (13). Equating the results we obtain

$$w(n,1)\,(1-p_1)\,p_1^n = (n+1)(1-r)^2 r^n \tag{18}$$

[1] Naturally, stochastic effects such as drift mean that in any finite population there is still random variation. However, in large populations these effects can be neglected.

since $C_{\mathcal{T}}(1, n) = 1$. The most natural match between r.h.s. and l.h.s. of (18) appears to be one where $p_1 = r$ and $w(n, 1) = (n + 1)(1 - p_1)$.

This last constraint completely rules out that $w(a, n)$ be constant, indicating that the length distribution under subtree crossover cannot be purely the result of a branching process (i.e., it is not Lagrangian). Instead, it suggests that $\Pr\{\ell\}$ is the product between the frequency provided by a branching process and the length ℓ of programs. So, we postulate that in general

$$w(a, n) = (an + 1)f(p_a) \tag{19}$$

where $f(p_a)$ is a function of p_a to be determined.

With this assumption, we impose (14), i.e., that probabilities sum to 1, obtaining:

$$f(p_a) = \frac{1}{\sum_{n\geq 0}(an + 1)C_{\mathcal{T}}(a, n)(1 - p_a)^{(a-1)n+1} p_a^n}. \tag{20}$$

The denominator of this equation is (by definition) $E[aN + 1] = \sum_n (an + 1)\Pr\{N = n\}$, where $\Pr\{N = n\}$ is given in (3). So, it is the expected length of the trees generated by a branching process where arity a nodes are used with probability p_a and arity 0 nodes used with probability $1 - p_a$. So, from (9) we have

$$f(p_a) = (1 - ap_a), \tag{21}$$

and so $w(a, n) = (an + 1)(1 - ap_a)$. As a result, we can now explicitly write the *tree-size distribution at the crossover fixed-point*:

$$\Pr\{n\} = (1 - ap_a)\binom{an + 1}{n}(1 - p_a)^{(a-1)n+1} p_a^n \tag{22}$$

where we used the explicit expression of $C_{\mathcal{T}}(a, n)$ in (4). This is the fixed-point tree-size distribution we were looking for. This distribution belongs to a family of distributions called *Lagrange distributions of the second kind* [9,8], which, until now, have never been related to branching processes and trees.

We can now impose constraint (15), i.e., equality of means, to infer the value of p_a:

$$\begin{aligned}
\mu_0 &= \sum_{n\geq 0}(an + 1)\Pr\{n\} \\
&= \sum_{n\geq 0}(1 - ap_a)(an + 1)^2 C_{\mathcal{T}}(a, n)\ (1 - p_a)^{(a-1)n+1} p_a^n \\
&= (1 - ap_a)\sum_{n\geq 0}(an + 1)^2 \Pr\{N = n\} \\
&= (1 - ap_a)E[L^2] \qquad \text{(by definition)},
\end{aligned}$$

and so, from (11),

$$\mu_0 = \frac{(a - 1)ap_a - a^2p_a^2 + 1}{(1 - ap_a)^2}. \tag{23}$$

By solving this equation for p_a we obtain

$$p_a = \frac{2\mu_0 + (a-1) - \sqrt{((1-a) - 2\mu_0)^2 + 4(1-\mu_0^2)}}{2a(1+\mu_0)} \tag{24}$$

which, encouragingly, for $a = 1$ collapses to the familiar $p_a = \frac{\mu_0 - 1}{\mu_0 + 1}$ (see [17]).

4 Evolution of the Tree-Size Distribution

Let us term *supertree* the part of a tree remaining after the removal of a subtree rooted at a particular crossover point. The size, L, of a tree after crossover is a stochastic variable obtained by adding the size X of a supertree randomly drawn from the population with the size Y of a subtree also randomly drawn from the population. I.e., $L = X + Y$. It follows that the probability distribution of L is the convolution of the subtree size distribution with the supertree size distribution. That is:

$$\Pr\{L = \ell\} = \sum_{i=0}^{\ell} \Pr\{X = i\} \Pr\{Y = \ell - i\} \tag{25}$$

For the case where internal nodes of arity a only are allowed, effectively a supertree always contains ja nodes and a subtree always contains $ka + 1$ nodes, where j and k are suitable non-negative integers. Therefore, (25) can be rewritten in terms of internal nodes. So, if N is the number of internal nodes of the tree, N_X the internal nodes in the supertree and N_Y the internal nodes in the subtree

$$\Pr\{N = n\} = \sum_{i=0}^{n} \Pr\{N_X = i\} \Pr\{N_Y = n - i\} \tag{26}$$

Naturally we have can interpret $\Pr\{N_X = i\}$ as a marginal and, so,

$$\Pr\{N_X = i\} = \sum_k \Pr\{N_X = i, N = k\} = \sum_k \Pr\{N_X = i | N = k\} \Pr\{N = k\} \tag{27}$$

where $\Pr\{N_X = i | N = k\}$ is the probability of extracting supertrees of size i from individuals of size k and $\Pr\{N = k\}$ is the distribution of tree sizes in the population. Of course, $\Pr\{N_X = i | N = k\} = 0$ for $k < i$, and so

$$\Pr\{N_X = i\} = \sum_{k \geq i} \Pr\{N_X = i | N = k\} \Pr\{N = k\} \tag{28}$$

We can similarly decompose $\Pr\{N_Y = n - i\}$ obtaining

$$\Pr\{N_Y = n - i\} = \sum_{k \geq n-i} \Pr\{N_Y = n - i | N = k\} \Pr\{N = k\} \tag{29}$$

Naturally, standard GP crossover is symmetric, and, therefore, the probability of extracting a *supertree* of size i from a tree of size k is identical to the probability of extracting a *subtree* of size $k-i$ from a tree of size k, i.e., $\Pr\{N_X = i|N = k\} = \Pr\{N_Y = k-i|N = k\}$. So, we have

$$\Pr\{N_X = i\} = \sum_{k \geq i} \Pr\{N_Y = k-i|N = k\} \Pr\{N = k\}. \tag{30}$$

By substituting (29) and (30) in (31) we finally obtain our *size-distribution evolution equation*:

$$\Pr\{n\}_{\text{new}} = \sum_{i=0}^{n} \sum_{k_1 \geq i} p(k_1 - i, k_1) \Pr\{k_1\} \sum_{k_2 \geq n-i} p(n-i, k_2) \Pr\{k_2\} \tag{31}$$

where we used the shorthand notation $p(a,b) = \Pr\{N_Y = a|N = b\}$ and $\Pr\{a\} = \Pr\{N = a\}$.

If the size distribution we propose in (22) is indeed the limiting distribution of sizes obtained by crossover in the absence of selection, then when one replaces $\Pr\{k_i\}$ with (22) in the r.h.s. of the previous equation, upon simplification one should obtain (22) again. This requires, however, the specification of the distribution of subtree sizes $p(a,b)$. In the next section we will obtain this distribution for the case where the limiting distribution of shapes follow the second assumption we used to derive (22), i.e., that within each class of tree sizes, all possible tree shapes are equally likely.

5 Subtree Distribution at the Crossover Fixed-Point

Let $s(n,k)$ be the expected number of subtrees with k internal nodes in trees with n internal nodes (where we draw trees of a particular size with uniform probability). Naturally, $s(n,n) = 1$. Also, under the assumption that only a-ary nodes and leaves can be used in the tree, we also have $s(n,0) = (a-1)n + 1$.

Let us consider all trees of size $n > 0$. These must all have a a-ary root node ($a > 0$). Let n_i be the size of child i of the root (naturally, $\sum_i n_i = n - 1$). Then we can divide up the space of trees of size n into groups based on the values of n_i. In each group there are $\prod_i C_{\mathcal{T}}(a, n_i)$ trees and, so, the probability of randomly drawing a tree belonging to a specific group when sampling trees of length n is given by $\frac{\prod_i C_{\mathcal{T}}(a,n_i)}{C_{\mathcal{T}}(a,n)}$. So, for example, the first group is characterised by $n_1 = 0$, $n_2 = 0$, $\cdots$, $n_a = n-1$ and contains $(C_{\mathcal{T}}(a,0))^{a-1} C_{\mathcal{T}}(a, n-1) = C_{\mathcal{T}}(a, n-1)$ trees. So, the probability of randomly obtaining a member of this group is $C_{\mathcal{T}}(a,n-1)/C_{\mathcal{T}}(a,n) = \frac{an+1}{a(n-1)+1}\binom{a(n-1)+1}{n-1}/\binom{an+1}{n}$.

Let $s_i(n_i,k)$ be the expected number of subtrees of size k for child i of the root. We assume that we know these quantities and we want to compute $s(n,k)$ on the basis of the $s_i(n_i,k)$. Clearly, for most values of k and n_i, $s(n,k)$ is simply going to be the sum of the $s_i(n_i,k)$'s, i.e., the number of trees of a given size in our tree is just the sum of the trees of that same size in all the children of the

root note. There are, however, special cases where we need to be more careful. In particular, if $k > n_i$, then $s_i(n_i, k) = 0$ and, therefore, $s_i(n_i, n) = 0$. However, $s(n, n) = 1$. So, there is one exception to the summation rule.

Below we will formalise the rule. However, before we do that, let us consider the effect of our assumption that within each length class all possible tree shapes happen with equal chance. This assumption leads to the fact that both trees and subtrees must follow the same subtree distribution, i.e., we have that $s_i(n_i, k) = s(n_i, k)$. Also, in order to compute $s(n, k)$ we need to sum over all possible ways in which we can draw the n_i's ensuring that the correct probability for each is considered. All this is accounted for in the following recursion:

$$s(n,k) = \begin{cases} 0 & \text{if } n < k, \\ 1 & \text{if } n = k, \\ \displaystyle\sum_{\sum n_i = n-1} \left(\frac{\prod_i C_{\mathcal{T}}(a, n_i)}{C_{\mathcal{T}}(a, n)} \right) \left(\sum_i s(n_i, k) \right) & \text{otherwise.} \end{cases} \tag{32}$$

So, the probability of drawing a tree with k internal nodes out of the class of trees of n nodes is given by

$$p(n,k) = \frac{s(n,k)}{\sum_k s(n,k)} = \frac{s(n,k)}{an+1} \tag{33}$$

since, rather obviously, $\sum_k s(n,k)$ is the total number of nodes in a tree with n internal nodes of arity a. As a result we can write

$$p(n,k) = \frac{\delta(k=n)}{an+1} + \delta(k<n) \sum_{\sum n_i = n-1} \left(\frac{\prod_i C_{\mathcal{T}}(a, n_i)}{C_{\mathcal{T}}(a, n)} \right) \sum_i \left(\frac{an_i + 1}{an+1} \right) p(n_i, k) \tag{34}$$

where $\delta(x) = 1$ if x is true, 0 otherwise.

6 Conjecture or Theorem?

In principle we now have all the ingredients to prove that (22) and the related distribution (34) are the fixedpoint for the tree-size-distribution evolution equation (31). However, the recursive nature of (34) and the complexities of simplifying sums of products of Catalan numbers make proving the result extraordinarily difficult (except for the case $a = 1$, since this leads directly to the result already proven in [12,15,13,14]).

To corroborate our result, we have therefore followed two alternative approaches. Firstly, we have collected empirical data on the size distributions obtained with different initialisations and for primitives of different arities in populations under the effect of crossover only. Secondly, we have performed a numerical integration of the r.h.s. of (31) at the assumed fixed point (for different values of a and μ_0) to verify if the resulting values for the l.h.s. matched the theoretical prediction. We describe the results of our tests in Sections 6.1 and 6.2.

6.1 Empirical Validation

We performed runs of a GP system in Java, with populations of 100,000 individuals, run for 500 generations. We used such large population sizes to reduce stochastic effects such as drift of the mean program size and to ensure that enough programs in each length-class were available. Similarly we performed a large number of generations to ensure that the initial conditions (see below) were completely washed out.

Only terminals and primitives of arity a (for $a = 1, 2, 3, 4, 5$) were allowed. Initialisation was performed using the "full" method. With this method, initial trees included $\mu_0 = d + 1$ primitives for $a = 1$ and $\mu_0 = \frac{1-a^{d+1}}{1-a}$ primitives for $a > 1$. Initial depth was 3, 4, or 8 (the root node being at depth 0).

During our runs we recorded histograms of program sizes, one for each generation, for sizes between 1 and 1000. Note that, because of the large population sizes and the particular objective of our runs (i.e., the study of size distributions), there was no need to collect data in multiple independent runs (as it is customary for other types of empirical studies).

In all cases the match between theoretical predictions and empirical data is striking. Compare, for example, the theoretical predictions and the empirical results shown in Figures 1–3.

6.2 Numerical Integration

The exact numerical integration of the r.h.s. of (31) at the assumed fixed point would require performing infinite sums, which is clearly impossible. So, we chose to limit sums over tree sizes to `limit` $= 5\mu_0$, effectively assuming that $\Pr\{n\} = 0$ for $n >$ `limit`. Naturally, this leads to some integration errors, but these turned out to be negligible for the purpose of confirming whether or not the resulting values for the l.h.s. matched the theoretical prediction.

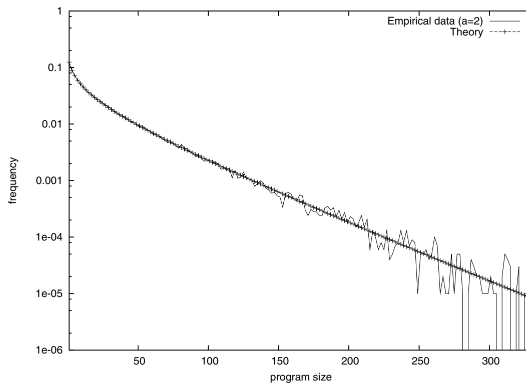


Fig. 1. Comparison between empirical and theoretical program size distributions for binary trees ($a = 2$) initialised with full method ($d = 4$, initial mean size $\mu_0 = 31$, mean size after 500 generations $\mu = 27.26044$)

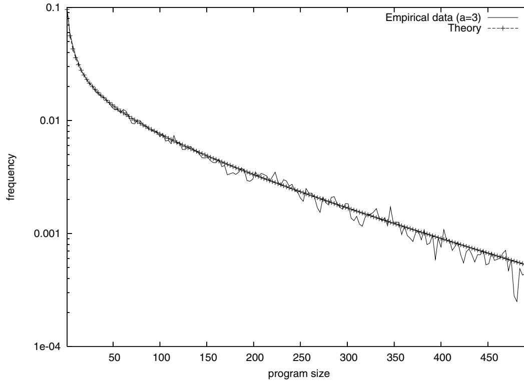


Fig. 2. Comparison between empirical and theoretical program size distributions for ternary trees ($a = 3$) initialised with full method ($d = 4$, initial mean size $\mu_0 = 121$, mean size after 500 generations $\mu = 109.10284$)

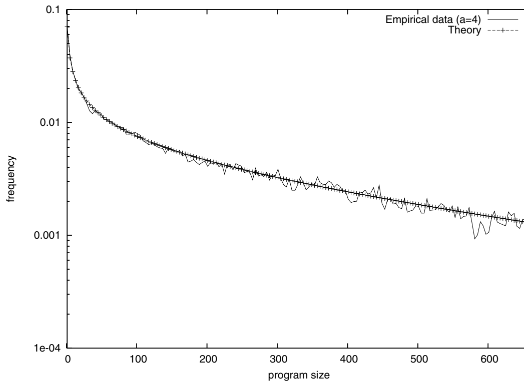


Fig. 3. Comparison between empirical and theoretical program size distributions for quaternary trees ($a = 4$) initialised with full method ($d = 4$, initial mean size $\mu_0 = 341$, mean size after 500 generations $\mu = 361.73052$)

We performed the integration for tree sizes (internal nodes) between 0 and `limit` inclusive, and for a variety of a's and μ_0's. In all cases, the output distribution computed via (31) was effectively indistinguishable from (22).

As an example of the degree of accuracy in the match between input and output size distributions, we show in Figure 4 a comparison between our conjectured program size distributions for binary trees and the output produced by (31). The plots of the distributions overlap almost perfectly. Indeed, absolute errors range between -2.8781×10^{-5} and -6.7263×10^{-7}, corresponding to relative errors between -0.023% and -0.049%.

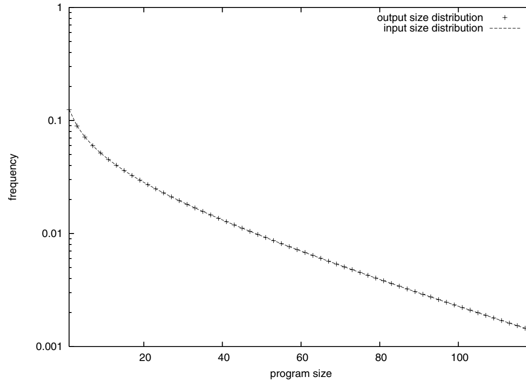


Fig. 4. Comparison between the program size distributions conjectured in (22) and output produced by (31) for binary trees ($a = 2$) of mean size $\mu = 27.26044$, as in Figure 1

7 Discussion and Conclusions

We have provided very strong theoretical and empirical evidence that the distribution of tree sizes towards which crossover pushes a population of unary, binary, ternary, etc. GP trees is a Lagrange distribution of the second kind. This result generalises results previously reported in [12,15,13,14].

Naturally, there are important consequences of this result. As was done in [12], we can now compute, for example, the expected resampling probability for programs of different sizes. In particular, let us imagine that our GP system operating on a flat landscape is at the fixed point distribution and let $\mathcal{F}$ and $\mathcal{T}$ be the sizes of the function and terminal sets, respectively. Since, there are $\mathcal{F}^n\mathcal{T}^{(a-1)n+1}$ different programs with n internal nodes in the search space, it is now possible to compute the average probability $p_{\text{sample}}(n)$ that each of these will be sampled by standard crossover, namely

$$p_{\text{sample}}(n) = \frac{(1 - ap_a)}{\mathcal{F}^n\mathcal{T}^{(a-1)n+1}} \binom{an+1}{n} (1 - p_a)^{(a-1)n+1} \, p_a^n. \tag{35}$$

It is easy to study this function and to conclude that, for a flat landscape, standard GP will sample a particular short program much more often than it will sample a particular long one. For example, when $\mu_0 = 27.26044$ as in Figures 1 and 4, GP will heavily resample short programs, e.g., the same program of length 1 is resampled on average every 16 crossovers, every 89 crossovers for programs of length 3, and every 448 crossover for length 5. However, as program size grows the sampling probability drops dramatically. For example, the resampling rate for programs of length 21 is 1 in over 77 million.

In future work we intend to extend the results reported here to the case where primitives of different arities are used in a run. Also, we will attempt to find a mathematical proof that (22) is a fixed-point for (31).

References

1. T. Blickle and L. Thiele. A mathematical analysis of tournament selection. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms (ICGA'95)*, pages 9–16, San Francisco, California, 1995. Morgan Kaufmann Publishers.
2. T. Blickle and L. Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, 1997.
3. P. C. Consul and L. R. Shenton. Use of Lagrange Expansion for Generating Discrete Generalized Probability Distributions. *SIAM Journal on Applied Mathematics*, 23(2):239–248, 1972.
4. H. Geiringer. On the probability theory of linkage in Mendelian heredity. *Annals of Mathematical Statistics*, 15(1):25–57, March 1944.
5. D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*. Morgan Kaufmann Publishers, 1991.
6. I. J. Good. The Lagrange Distributions and Branching Processes. *SIAM Journal on Applied Mathematics*, 28(2):270–275, 1975.
7. P. Hilton and J. Pederson. Catalan numbers, their generalization, and their uses. *Mathematical Intelligencer*, 13:64–75, 1991.
8. K. Janardan. Weighted Lagrange Distributions and Their Characterizations. *SIAM Journal on Applied Mathematics*, 47(2):411–415, 1987.
9. K. Janardan and B. Rao. Lagrange Distributions of the Second Kind and Weighted Distributions. *SIAM Journal on Applied Mathematics*, 43(2):302–313, 1983.
10. S. Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283, Sept. 2000.
11. T. Motoki. Calculating the expected loss of diversity of selection schemes. *Evolutionary Computation*, 10(4):397–422, 2002.
12. R. Poli and N. F. McPhee. Exact schema theorems for GP with one-point and standard crossover operating on linear structures and their application to the study of the evolution of size. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 126–142, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.
13. R. Poli and N. F. McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part I. *Evolutionary Computation*, 11(1):53–66, Mar. 2003.
14. R. Poli and N. F. McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part II. *Evolutionary Computation*, 11(2):169–206, June 2003.
15. J. E. Rowe and N. F. McPhee. The effects of crossover and mutation operators on variable length linear structures. Technical Report CSRP-01-7, University of Birmingham, School of Computer Science, Jan. 2001.
16. M. D. Vose. *The simple genetic algorithm: Foundations and theory.* MIT Press, Cambridge, MA, 1999.
17. H. W. Watson and F. Galton. On the Probability of the Extinction of Families. *The Journal of the Anthropological Institute of Great Britain and Ireland*, 4:138–144, 1875.

Predicting Prime Numbers Using Cartesian Genetic Programming

James Alfred Walker and Julian Francis Miller

Intelligent Systems Group, Department of Electronics, University of York,
Heslington, York, YO10 5DD, UK
{jaw500,jfm7}@ohm.york.ac.uk

Abstract. Prime generating polynomial functions are known that can produce sequences of prime numbers (e.g. Euler polynomials). However, polynomials which produce consecutive prime numbers are much more difficult to obtain. In this paper, we propose approaches for both these problems. The first uses Cartesian Genetic Programming (CGP) to directly evolve integer based prime-prediction mathematical formulae. The second uses multi-chromosome CGP to evolve a digital circuit, which represents a polynomial. We evolved polynomials that can generate 43 primes in a row. We also found functions capable of producing the first 40 consecutive prime numbers, and a number of digital circuits capable of predicting up to 208 consecutive prime numbers, given consecutive input values. Many of the formulae have been previously unknown.

1 Introduction

There are many questions relating to properties of primes numbers that have fascinated mathematicians for hundreds of years [1]. It is well known that no formulae have ever been produced that can map the sequence of natural numbers into the sequence of primes. However there exists many simple polynomials that can map quite long sequences of natural numbers into a sequence of distinct primes (long is generally measured with respect to the so-called Euler's polynomial $x^2 - x + 41$ [2] which produces distinct primes for values of x from $1 \leq x \leq 40$). Euler's polynomial continues to produce many primes for larger values of x.

Legendre found a similar polynomial $x^2 + x + 41$ which produces prime numbers for $0 \leq x \leq 39$, and it is this polynomial which, oddly, is referred to as *Euler's polynomial* [3,1]. In this paper we present evolved polynomials that can generate long sequences of primes (including re-discovering Euler's and Legendre's polynomials). Recently there has been renewed interest in the mathematics of prime-producing polynomials [4]. In evaluating the quality of prime-producing polynomials we must observe that there can be many criteria for deciding on the fecundity of prime-producing polynomials. Since polynomials can produce positive or negative quantities, some discovered polynomials are particularly fecund at generating positive or negative primes. Other polynomials can produce long sequences of primes, however prime values may be repeated. The most sought

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 205–216, 2007.
© Springer-Verlag Berlin Heidelberg 2007

after quality of prime-producing polynomials appears to be the longest sequence of positive distinct primes [5]. Last year there were many computational attempts at producing prime-producing polynomials and a polynomial of degree five was found to be particularly good (though not at producing distinct, positive primes) [6].

Ulam discovered that there are many integer coefficients, b and c, such that $4x^2 + bx + c$ generates a large number of prime numbers [1]. The polynomial $41x^2 + 33x + 43321$ has also been shown to produce prime numbers for ninety value of x, when $0 \leq x \leq 99$, but only twenty six of the primes are consecutive [1]. A high asymptotic density of primes is often considered to be an important criterion of the fecundity of prime-producing polynomials [7]. Gilbert Fung announced his discovery of two polynomials $103x^2 - 3945x + 34381$ and $47x^2 - 2247x + 21647$ which produces prime numbers for $0 \leq x \leq 43$. However, the best polynomial found so far is $36x^2 - 810x + 2753$, which was discovered by Ruby (immediately after hearing Fung's announcement), and produces primes numbers for $0 \leq x \leq 44$ [5]. The interested reader may consult [8] and [9] for more recent mathematical findings on the subject. Since polynomials of fixed order can be easily transformed by translation operations, there are in fact infinitely many quadratics that have 'Euler-fecundity'. The most important mathematical quantity characterising the essential behaviour of prime-producing polynomials is the polynomial discriminant which for a quadratic of form $ax^2 + bx + c$ is $b^2 - 4ac$. Mollin gives tables of polynomials with particular discriminants that produce long sequences of primes [5].

Euler's polynomial was the inspiration behind one of the GECCO competitions in 2006. The aim of the GECCO Prime Prediction competition (and some of the work in this paper) was to produce a polynomial $f(i)$ with integer coefficients, such that given an integer input, i, it produces the ith prime number, $p(i)$, for the largest possible value of i. For example, $f(1) = 2$, $f(2) = 3$, $f(3) = 5$, $f(4) = 7$. Therefore, the function $f(i)$ must produce consecutive prime numbers for consecutive values of i. The requirement that the polynomial must not only produce primes for consecutive input values, but also that the primes themselves must be consecutive, makes the problem considerably more challenging than mathematicians have previously considered. The two approaches described in Section 4.2 were entered in the GECCO Prime Prediction competition and were ranked second overall. The winning entry evolved floating point co-efficients of a polynomial using a Genetic Algorithm (GA), where the output of the polynomial was rounded to produce the prime numbers for consecutive values of i. However, the winning entry was only able to predict correctly a few consecutive prime numbers (9 in total). Unfortunately, the details regarding this have not been published.

So far, it seems that no integer polynomial exists, which is capable of producing sequences of consecutive prime numbers. In this paper, we are proposing two approaches to evolve a formula (in one case strictly a polynomial) capable of producing prime numbers. The first approach treats the consecutive prime number producing formula as a symbolic regression problem. The technique

used for these approaches is Cartesian Genetic Programming (CGP)[10]. The second approach evolves a digital circuit, which can produce consecutive prime numbers for consecutive input values. Any digital circuit can be represented as a polynomial expression, as any logic function can be expressed using only addition, subtraction or multiplication. The technique used to evolve the consecutive prime generating digital circuit is an extension of the CGP technique, known as multi-chromosome CGP [11]. Multi-chromosome CGP has been shown to significantly improve performance on difficult, multiple-output, digital circuit problems, when compared with the conventional form of CGP [11].

The discovery of new prime producing formulae (consecutive, or otherwise) would be of interest to mathematicians, as it is unknown whether such formulae currently exist. Even if such formulae do exist, they may be too complex for a human mathematician to discover. Therefore, this paper once again highlights the use of evolutionary computation as a tool for discovery and design. Also, we propose that the evolution of prime producing formulae would make an interesting and challenging benchmark for comparing evolutionary computation techniques, as it proved clear by empirical tests that it is a harder and more complex problem to solve than many existing GP benchmarks.

The plan for the paper is as follows: section 2 gives an overview of the CGP technique, followed in section 3 by a description of the multi-chromosome extension to the CGP technique. The details of our experiments on evolving sequences of prime numbers are shown in section 4, followed by the results in section 5. Section 6 gives conclusions and some suggestions for future work.

2 Cartesian Genetic Programming (CGP)

Cartesian Genetic Programming is a form of Genetic Programming (GP) invented by Miller and Thomson [10], for the purpose of evolving digital circuits. However, unlike the conventional tree-based GP [12], CGP represents a program as a directed graph (that for feed-forward functions is acyclic). The benefit of this type of representation is that it allows the implicit re-use of nodes in the directed graph. CGP is also similar another technique called Parallel Distributed GP, which was independently developed by Poli [13]. Originally CGP used a program topology defined by a rectangular grid of nodes with a user defined number of rows and columns. However, later work on CGP showed that it was more effective when the number of rows is chosen to be one [14]. This one-dimensional topology is used throughout the work we report in this paper.

In CGP, the genotype is a fixed length representation and consists of a list of integers which encode the function and connections of each node in the directed graph. However, the number of nodes in the program (phenotype) can vary but is bounded, as not all of the nodes encoded in the genotype have to be connected. This allows areas of the genotype to be inactive and have no influence on the phenotype, leading to a neutral effect on genotype fitness called neutrality. This unique type of neutrality has been investigated in detail and found to be extremely beneficial to the evolutionary process on the problems studied [10,15,14].

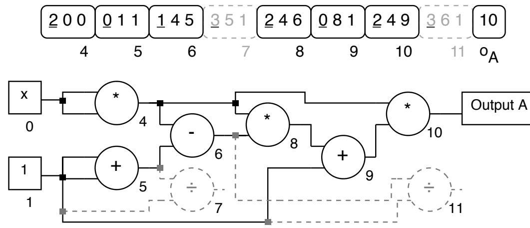


Fig. 1. A CGP genotype and corresponding phenotype for the function $x^6 - 2x^4 + x^2$. The underlined genes in the genotype encode the function of each node, the remaining genes encode the node inputs. The function lookup table is: $+(0)$, $-(1)$, $*(2)$, $\div(3)$. The index labels are shown underneath each program input and node. The inactive areas of the genotype and phenotype are shown in grey dashes.

Each node is encoded by a number of genes. The first gene encodes the node function, whilst the remaining genes encode where the node takes its inputs from. The nodes take their inputs in a feed forward manner from either the output of a previous node or from the program inputs (terminals). Also, the number of inputs that a node has is dictated by the arity of its function. The program inputs are labelled from 0 to n-1, where n is the number of program inputs. The nodes encoded in the genotype are also labelled sequentially from n to n+m-1, where m is the user-defined bound for the number of nodes. If the problem requires k program outputs, then k integers are added to the end of the genotype, each encoding a node output in the graph where the program output is taken from. These k integers are initially set as the outputs of the last k nodes in the genotype. Fig. 1 shows a CGP genotype and corresponding phenotype for the function $x^6 - 2x^4 + x^2$ and Fig. 2 shows the decoding procedure.

3 Multi-chromosome Cartesian Genetic Programming

3.1 Multi-chromosome Representation

The difference between a CGP genotype (described earlier in section 2) and a Multi-chromosome CGP genotype, is that the Multi-chromosome CGP genotype is divided into a number of equal length sections called chromosomes. The number of chromosomes present in a genotype is dictated by the number of program outputs required by the problem, as each chromosome is connected to a *single* program output. This allows large problems with multiple-outputs (normally encoded in a single genotype), to be broken down into many smaller problems (each encoded by a chromosome) with a single output. This approach should make the larger problems easier to solve. By allowing each of the smaller problems to be encoded in a chromosome, the whole problem is still encoded in a single genotype but the interconnectivity between the smaller problems (which can cause obstacles in the fitness landscape) has been removed.

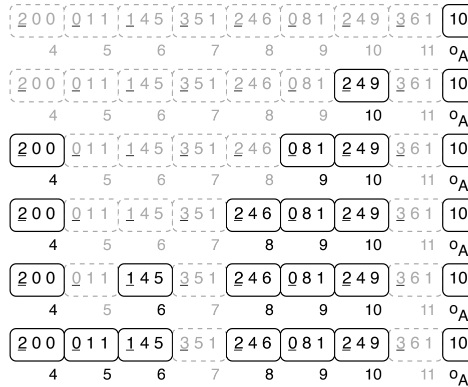


Fig. 2. The decoding procedure of a CGP genotype for the function $x^6 - 2x^4 + x^2$. a) Output A (o_A) connects to the output of node 10, move to node 10. b) Node 10 connects to the output of nodes 4 and 9, move to nodes 4 and 9. c) Nodes 4 and 9 connect to the output of node 8 and program inputs 0 and 1, move to node 8. d) Node 8 connects to the output of nodes 4 and 6, move to node 6, as node 4 has already been decoded. e) Nodes 6 connects to the output of nodes 4 and 5, move to node 5. f) Node 5 connects to program input 1. When the recursive process has finished, the genotype is fully decoded.

Each chromosome contains an equal number of nodes, and is treated as a genotype of an individual with a single program output. The inputs of each node encoded in a chromosome are only allowed to connect to the output of earlier nodes encoded in the same chromosome or any program input (terminals). This creates a form of compartmentalization in the genotype which supports the idea of removing the interconnectivity between the smaller problems encoded in each chromosome. An example of a Multi-chromosome CGP genotype is shown in Fig. 3.

3.2 Fitness Function and Multi-chromosome Evolutionary Strategy

The fitness function used in multi-chromosome approach is identical to the fitness function used in single chromosome approach, except for one small change. The output of each chromosome in multi-chromosome approach is calculated and assigned a fitness value based on the hamming distance from the perfect solution of a single output, whereas in CGP a fitness values is assigned to the whole genotype based on the hamming distance from the perfect solution over all the outputs (a perfect solution has a fitness of zero). Therefore, the multi-chromosome approach has n fitness values, where n is the number of program outputs, per genotype. This allows each chromosome in a genotype to be compared with the corresponding chromosome in other genotypes, by using a (1 + 4) multi-chromosome evolutionary strategy.

The (1 + 4) multi-chromosome evolutionary strategy selects the best chromosome at each position from all of the genotypes and generates a new best

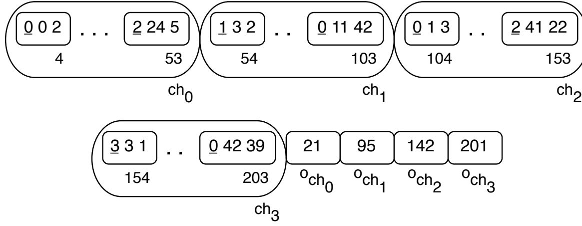


Fig. 3. A Multi-chromosome CGP genotype with four inputs, four outputs ($o_{c_0} - o_{c_3}$) and four chromosomes ($c_0 - c_3$), each consisting of fifty nodes

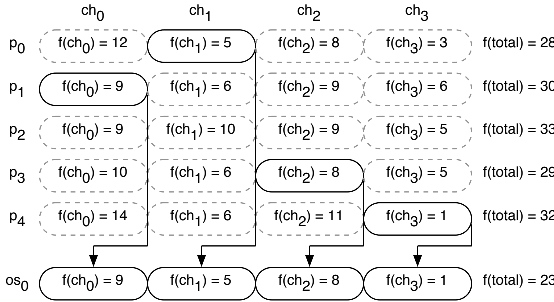


Fig. 4. The (1 + 4) multi-chromosome evolutionary strategy used in Multi-chromosome CGP. $p_{x,g}$ - parent x at generation g, c_y - chromosome y, $f(p_{x,g}, c_y)$ - fitness of chromosome y in parent x at generation g, $f(p_{x,g})$ - fitness of parent x at generation g.

of generation genotype containing the fittest chromosome at each position. The new best of generation genotype may not have existed in the population, as it is a combination of the best chromosomes from all the genotypes, so it could be thought of as a "super" genotype. The multi-chromosome version of the (1 + 4) evolutionary strategy therefore behaves as an intelligent multi-chromosome crossover operator, as it selects the best parts from all the genotypes. The overall fitness of the new genotype will also be better than or equal to the fitness of any genotype in the population from which it was generated. An example of the multi-chromosome evolutionary strategy is shown in Fig. 4.

4 Evolving a Prime Producing Formulae

4.1 Non-consecutive Prime Producing Formulae

The approach chosen for attempting to evolve integer coefficient polynomials (e.g. Euler's) was to assume that the polynomial was quadratic in the index value with a CGP genotype corresponding to each coefficient. Each genotype took the index value i as the only input. The primitive functions used were integer addition, subtraction, multiplication, protected division, and protected

modulus. The CGP genotype was 300 primitives. One percent of all genes were mutated to create the offspring genotypes in a 1+4 evolutionary strategy (in which if any offspring were as fit at the best and there were no fitter genotypes, the offspring was always chosen). One hundred runs of 20,000 generations were carried out. The fitness of the polynomial encoded in the genotype was calculated by adding one for every true prime generated (for index values 0 to 49) that was bigger than the previous prime generated.

4.2 Consecutive Prime Producing Formulae

The aim of this experiment is to evolve a function $f(i)$, which is capable of producing consecutive prime numbers $p(i)$ for consecutive values of i. For example, $f(1) = 2$, $f(2) = 3$, $f(3) = 5$, $f(4) = 7$, etc. In this paper, we propose two approaches to evolving the polynomial $f(i)$; one treats $f(i)$ as an integer based function, while the other treats $f(i)$ as a binary based function.

An Integer Based Approach to the Prime Producing Polynomial. The first approach discussed uses CGP in a similar manner found in any symbolic regression approach [16]. The input of the CGP program is the i value, in the form of an integer, and the program output is the predicted prime number, $p(i)$, in the form of an integer. The function set used is similar to that used in many symbolic regression problems, comprising of addition, subtraction, multiplication, protected division and protected modulus. The CGP genotype is allowed 200 nodes, which can represent any of the functions in the function set. The fitness function used awards a point for every number produced which is a prime number and is in the correct consecutive position for the first 40 consecutive prime numbers.

A Binary Based Approach to the Prime Producing Polynomial. The second approach treats the polynomial $f(i)$, as a digital circuit problem, and uses multi-chromosome CGP to evolve a solution. Technically, the evolved solution will still be a polynomial, as any logical expression can be expressed in terms of a number of variables and the operators addition, subtraction and multiplication. Also, any input value i, when represented as a binary number, also forms a polynomial, $i = \sum a_j 2^j = a_n 2^n + a_{n-1} 2^{n-1} + ... + a_0 2^0$, where $0 \leq j \leq n$. Likewise, any prime number, $p(i)$, produced can also be represented as a binary number, and also forms a polynomial, $p(i) = \sum b_k 2^k = b_m 2^m + b_{m-1} 2^{m-1} + ... + b_0 2^0$, where $0 \leq k \leq m$. Therefore, we are trying to evolve a function $f(i)$, which given the coefficients of the binary number representing i, $a_0, ..., a_n$, produces the coefficients of the binary number representing $p(i)$, $b_0, ..., b_m$, where n does not have to equal m. An illustration of the process is shown in Fig. 5.

The function $f(i)$, which maps the coefficients of the input i to the output $p(i)$, is evolved using multi-chromosome CGP. The evolved program has n program inputs and m program outputs. In this case, $n = 14$, as this is the minimum number of inputs required to accept the number 10,000 in binary format and $m = 17$, as this is the minimum number of outputs required to produce the

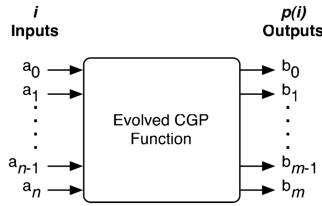


Fig. 5. The function mapping between the coefficients of the binary number representing the input, i, and the coefficients of the binary number representing the prime number output, $p(i)$

10,000th prime number. Each program output is taken from a separate chromosome in the genotype, therefore the genotype consists of m chromosomes. Each chromosome is an equal length and contains 300 nodes. The function set for the experiment simply contains a multiplexer which can choose either input in_0 or input in_1, as its output. The mutation rate used was 3% per chromosome.

As the set of test cases supplied for the GECCO competition was very large (10,000), and there was no guarantee a solution exists for all 10,000 test cases, or how much computational power would be required to find a solution, an incremental form of evolution was used. The evolved program starts off trying to find a solution to the first 16 test cases. If a solution is found, the run continues but the number of test cases is increased to 32. This evolutionary process continues, incrementing the number of test cases by 16 each time a solution is found, until a solution is found for all 10,000 test cases (a total of 625 increments).

In this paper, we are not actually benchmarking the performance of any of the techniques but we are using them for exploratory purposes, to see if any function can be discovered that is capable of predicting consecutive prime numbers.

5 Results and Discussion

5.1 Non-consecutive Prime Producing Polynomials

In the hundred runs, we obtained 6 Legendre polynomials and 5 Euler polynomials. The most common polynomial found was $2x^2 + 40x + 1$. This was found 57 times. The polynomial produces 47 primes for index values 0 to 49 but 17 is the longest sequence of primes. The most interesting solution obtained was the polynomial $x^2 - 3x + 43$. This produces primes for index values 0 to 42. This is a sequence of primes that is two primes longer than Euler or Legendre's polynomials. However, it has two repeats (the sequence begins 43, 41, 41, 43, 47, for index values 0,1,2,3,4). We could not find this polynomial in the literature (despite its simple form). When the number of generations was increased we found that the technique tended to converge on Euler of Legendre polynomials with much greater frequency (i.e. these polynomials are great 'attractors').

Further work was carried out in which polynomials were rewarded for having as large a sum of coefficients as possible (provided that they were equally good

at producing long sequences of primes). We carried out 1000 runs of 40,000 generations with 200 primitives in each coefficient producing program (quadratics). The inputs to the coefficient producing programs were chosen to be 19, 47, 97, 139, and 193 respectively. The Euler polynomial was produced 142 times and the polynomial $2x^2 + 40x + 1$ (second best) was discovered 14 times. This approach was found to produce a much greater variety of polynomials, many of which produced long sequences of primes. Some examples are $8x^2 + 104x + 139$ (25) and $2848x^2 + 73478x + 227951$ (15), where the figures in brackets represent the length of the sequence of primes produced.

5.2 The Integer-Based Approach

The symbolic regression approach, was run independently ten times for 100,000 generations. The results of these runs can be shown in Table 1. ¿From the results, the best individual run was picked with a fitness of 27 out of the first 40 primes correct. This individual was evolved for a further 10 million generations, by which it had reached a fitness of 37 after 3,192,104 generations. Once again, the individual was evolved for a further 20 million generations. This time it had now reached a fitness of 39 after 16,336,784 generations. The individual still had not found all 40 consecutive prime numbers, so it was evolved further until it could correctly produce the first 40 prime numbers consecutively, which took a further 48,755,397 generations. The solution contained 88 active nodes out of the original 200 nodes and required 113,176,917 potential solutions to be evaluated in order to find this solution, indicating the difficulty of this problem.

As an extension to the experiment, the evolved solution was evaluated on the first 100 prime numbers (60 of which it had never been trained on) to see how well the solution generalised. The evolved solution found 21 prime numbers out of the 60 prime numbers it had never seen before. Some of the prime numbers found in the 21 prime numbers were in small groups whilst others were spread out. This indicates that the evolved solution not only found the first 40 consecutive prime numbers but also learnt something about what it means to be a prime number.

Table 1. The results of 10 independent runs of CGP trying to find the first 40 consecutive prime numbers

Run No.	Final Fitness	Generation Achieved	No. Active Nodes
0	16	8257	48
1	17	5666	36
2	13	2331	37
3	16	4234	34
4	17	4955	37
5	19	6261	42
6	16	3447	41
7	27	9944	57
8	18	6383	57
9	15	7305	52

5.3 The Binary-Based Approach

The digital circuit approach was run continuously, incrementing the number of test cases each time a solution was found. Evolved solutions were found for the first 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192 and 208 consecutive prime numbers. The evolved solution that produces the first 16 consecutive primes is shown in Equation 1.

$$p(i) = b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0 \ where \tag{1}$$

$$\begin{aligned}
b_5 &= a_0 + a_1 a_2 - 2a_0 a_1 a_2 \\
b_4 &= -((a_1(2a_2 - 1) - a_2)(1 + a_2(a_3 a_4 - 1))) \\
&\quad + a_0(1 - 2a_2 + a_2^2(2 - 2a_3 a_4) + 2a_1(2a_2 - 1)(1 + a_2(a_3 a_4 - 1))) \\
b_3 &= a_2(a_3 + a_4 - 2a_3 a_4) + a_1 a_3(1 + a_2(2a_3 a_4 - a_3 - a_4)) \\
b_2 &= 1 - a_3 - a_4 + 2a_3 2a_4 - 2a_2^4(2a_3 - 1)3(a_4 - 1)a_4 \\
&\quad + 2a_3 a_4^2 - 2a_3^2 a_4^2 - a_2^2(2a_3 - 1)(a_3(2 - 6a_4) \\
&\quad + (3 - 2a_4)a_4 + 6a_3^2(a_4 - 1)a_4) + a_2^3(1 - 2a_3)^2(1 - (1 + 6a_3)a_4 \\
&\quad + (6a_3 - 2)a_4^2) + a_2(2a_3^3(a_4 - 1)a_4 + 2a_4^2 + a_3(2 + 5a_4 - 8a_4^2) \\
&\quad + a_3^2(1 - 9a_4 + 6a_4^2) - 1) + a_1(1 - a_2 a_3 + a_2^2(2a_3 - 1))(2a_3 \\
&\quad + 2a_4 - 1 - a_3 a_4 - 2a_3^2 a_4 + 2a_2^4(2a_3 - 1)^3(a_4 - 1)a_4 - 2a_3 a_4^2 \\
&\quad + 2a_3^2 a_4^2 + 2a_2^2(2a_3 - 1)(a_3(2 - 4a_4) - (a_4 - 2)a_4 \\
&\quad + 3a_3^2(a_4 - 1)a_4) - 2a_2^3(1 - 2a_3)^2(1 - (1 + 3a_3)a_4 + (3a_3 - 1)a_4^2) \\
&\quad - a_2(a_4 - 2 + 2a_3^3(a_4 - 1)a_4 + 2a_4^2 + a_3(4 + 4a_4 - 8a_4^2) \\
&\quad + 2a_3^2(1 - 5a_4 + 3a_4^2))) \\
b_1 &= 1 - a_3 + a_3^2 - a_2 a_3^2 - a_1(a_2(1 + a_3^2 + a_3(a_4 - 3)) \\
&\quad + a_3^2(1 - 2a_4) + a_2^2 a_3(2a_3 - 1)(a_4 - 1)) - a_3^2 a_4 + a_2 a_3^2 a_4 \\
&\quad + a_1^2(a_2 - 1)a_2 a_3(2a_3 - 1)(2a_4 - 1) \\
&\quad - a_0(2a_1 a_2 - 1)(a_3 - 1)(1 + (a_2 - 1)a_3(1 - a_4 + a_1(2a_4 - 1))) \\
b_0 &= a_2 - a_0(a_1 - 1)(a_2 - 1)(a_3 - 1) + a_1(a_2 - 1)(a_3 - 1) + a_3 - a_2 a_3
\end{aligned}$$

The solution producing 208 consecutive primes contained 400 active nodes and required 230,881,977 generations. A total of 923,527,909 potential solutions had to be evaluated, which required approximately three weeks of computing time on a PC with a single 1.83GHz processor and 448MB RAM. We believe that with enough computing power it would be possible to find a solution capable of predicting the first 10,000 prime numbers.

On examining the solutions, it can be observed that the more consecutive primes a solution can predict, the more active nodes the solution contains. The majority of the evolved solutions could not be included in this paper, as they were to large to print. Due to the sheer complexity of the solutions, we believe that it is highly unlikely that a human would ever devise such a solution, especially for the solutions producing high numbers of consecutive primes.

As the evolved solution for the first 16 prime numbers was capable of accepting inputs up to 31, we decided to extend the experiment to see how the solution generalised on 15 previously unseen inputs (just as we did with the integer-based approach). From the 15 unseen inputs, 7 of the predicted 15 outputs were prime numbers, which is just below 50%, indicating that the solution had learned something about "primeness" or favoured prime numbers. However, none of the 7 prime numbers produced from the 15 unseen inputs were consecutive.

6 Conclusion and Future Work

In this paper, we have presented an approach for evolving non-consecutive prime generating polynomials and also two different approaches using CGP for evolving a function $f(i)$, which produces consecutive prime numbers $p(i)$, for consecutive input values i. The best non-consecutive prime generating polynomial evolved produced 43 primes in a row (better than Euler's). Of the consecutive prime generating formulae, the symbolic regression approach using CGP, evolved a function capable of producing 40 consecutive prime numbers for input value i, where $1 \leq i \leq 40$. The digital circuit approach using multi-chromosome CGP, evolved multiple functions for consecutive sequences of prime numbers with increasing length, the longest of which produced 208 consecutive prime numbers, for input value i, where $1 \leq i \leq 208$. Although the second approach produced much larger sequences of prime numbers, the size of the solutions were enormous, in comparison with those produced by the first approach. In future work, once a solution is found, we intend to continue the evolutionary process with an altered fitness function, which minimises the number of nodes used. Therefore, making the solutions more compact. The downside of this approach is any generality evolved for solving further test cases could be lost.

The binary approach produced larger numbers of consecutive primes much easier than the integer-based approach, possibly indicating that by altering the search space from $\log_{10}$ to $\log_2$ has discovered a previously unknown relationship between the prime numbers. It is possible that by investigating other bases in the future, such as $\log_8$ or $\log_{16}$ could produce further links between prime numbers and help in discovering a function for prime prediction.

References

1. Wells, D.: Prime Numbers. John Wiley and sons (2005)
2. Euler, L.: Extrait d'un lettre de m. euler le pere a m. bernoulli concernant le memoire imprime parmi ceux de 1771. Nouveaux Mémoires de l'Académie royale des Sciences de Berlin, Histoire (1772) 35–36
3. Legendre, A.M.: Théorie des nombres. 2 edn. Libraire Scientifique A. Herman (1808)
4. Mollin, R.: Quadratics. Boca Raton (1995)
5. Mollin, R.: Prime-producing quadratics. American Mathematical Monthly **104**(6) (1997) 529–544

6. Pegg Jnr., E.: Math games: Prime generating polynomials
7. Fung, G., Williams, H.: Quadratic polynomials which have a high density of prime values. Mathematics of Computation **55** (1990) 345–353
8. Mollin, R.: New prime-producing quadratic polynomials associated with class number one or two. New York Journal of Mathematics **8** (2002) 161–168
9. Harrell, H.: Prime Producing Equations: The Distribution of Primes and Composites Within a Special Number Arrangement. AuthorHouse (2002)
10. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Proceedings of the 3rd European Conference on Genetic Programming (EuroGP 2000). Volume 1802 of LNCS., Edinburgh, UK, Springer-Verlag (15-16 April 2000) 121–132
11. Walker, J.A., Miller, J.F.: A multi-chromosome approach to standard and embedded cartesian genetic programming. In: Proceedings of the 2006 Genetic and Evolutionary Computation Conference (GECCO 2006). Volume 1., Seattle, Washington, USA, ACM Press (8-12 July 2006) 903–910
12. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)
13. Poli, R.: Parallel Distributed Genetic Programming. Technical Report CSRP-96-15, School of Computer Science, University of Birmingham, B15 2TT, UK (September 1996)
14. Yu, T., Miller, J.F.: Neutrality and the evolvability of boolean function landscape. In: Proceedings of the 4th European Conference on Genetic Programming (EuroGP 2001). Volume 2038 of Lecture Notes in Computer Science., Springer-Verlag (2001) 204–217
15. Vassilev, V.K., Miller, J.F.: The advantages of landscape neutrality in digital circuit evolution. In: Proceedings of the 3rd International Conference on Evolvable Systems (ICES 2000). Volume 1801 of Lecture Notes in Computer Science., Springer Verlag (2000) 252–263
16. Walker, J.A., Miller, J.F.: The automatic acquisition, evolution and re-use of modules in cartesian genetic programming. Accepted for publication in IEEE Transactions on Evolutionary Computation

Real-Time, Non-intrusive Evaluation of VoIP

Adil Raja[1], R. Muhammad Atif Azad[2], Colin Flanagan[1], and Conor Ryan[2]

[1] Wireless Access Research Center
Department of Electronic and Computer Engineering
[2] Biocomputing and Developmental Systems Group
Department of Computer Science and Information Systems
University of Limerick, Limerick, Ireland
{adil.raja,atif.azad,colin.flanagan,conor.ryan}@ul.ie

Abstract. Speech quality, as perceived by the users of Voice over Internet Protocol (VoIP) telephony, is critically important to the uptake of this service. VoIP quality can be degraded by network layer problems (delay, jitter, packet loss). This paper presents a method for real-time, non-intrusive speech quality estimation for VoIP that emulates the *subjective* listening quality measures based on *Mean Opinion Scores* (MOS). MOS provide the numerical indication of perceived quality of speech. We employ a Genetic Programming based symbolic regression approach to derive a speech quality estimation model. Our results compare favorably with the International Telecommunications Union-Telecommunication Standardization (ITU-T) PESQ algorithm which is the most widely accepted standard for speech quality estimation. Moreover, our model is suitable for real-time speech quality estimation of VoIP while PESQ is not. The performance of the proposed model was also compared to the new ITU-T recommendation P.563 for non-intrusive speech quality estimation and an improved performance was observed.

Keywords: VoIP, Non-Intrusive, Speech Quality, GP, Symbolic Regression, MOS.

1 Introduction

Speech quality estimation for VoIP can be performed either *subjectively* or *objectively*. In the former case, speech quality is estimated by averaging the opinions of a set of suitably trained human subjects [1]. Each of the testers assigns an *Opinion Score* – on an integral scale from 1 (unacceptable) to 5 (excellent) – to the speech signal under test. The opinion scores of the testers are averaged into a MOS. Subjective MOS has been found to be, by far, the most reliable technique of speech quality estimation. However, it is expensive, time-consuming and laborious.

Recently, objective speech quality assessment has become a very active research area. This is an attempt to circumvent the limitations of subjective testing by simulating the opinions of human testers algorithmically. There are two distinct approaches to objective testing: intrusive and non-intrusive.

Intrusive speech quality estimation techniques compare the test (i.e., network distorted) speech signal, as reconstructed by the decoder, to the reference, input

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 217–228, 2007.
© Springer-Verlag Berlin Heidelberg 2007

speech, basing their estimation on the measured amount of distortion. ITU-T P.862 (PESQ) [2] is a popular example of intrusive estimation model.

On the other hand non-intrusive schemes assess the quality of the distorted signal in the absence of the reference signal. This approach is effective in environments where the reference speech signal is not accessible. P.563 is the new ITU-T Recommendation for non-intrusive evaluation speech quality in narrow-band telephony applications [3]. Intrusive models are more reliable than the non-intrusive ones as the former have access to a reference speech signal to compare the distorted speech signal with.

However, the afore-mentioned models are compute-intensive as they base their results on the time and/or frequency domain analysis of the speech signal under test. They also require the test call to be recorded for a considerable duration before it can be analysed. Hence, they are not suitable for real-time and continuous monitoring of speech quality. This makes non-intrusive models like ITU-T Recommendation G.107 (E-Model) [4] more attractive for real-time speech quality estimation as they base their results on networks traffic parameters. Despite the fact that E-model is a *transmission planning* tool, it has been deployed in various commercial applications. First of all a different version of it exists for various network conditions such as codec type and bursty or non-bursty network conditions. Moreover, it is restricted to a limited number of codecs and network conditions due to its reliance on subjective tests [5].

In this paper we have employed Genetic Programming (GP) based symbolic regression approach to estimate the speech quality as a function of impairments due to IP network and encoding algorithms. A main advantage of GP is that it can produce human-readable results in the form of analytical expressions. Moreover, GP deals with the significant input parameters and aids in the automatic pruning of the irrelevant ones. These features of GP make our results superior to the past research based on Artificial Neural Networks (ANNs) by Sun and Ifeachor [6], Mohamed et. al. [7] [8] and on lookup tables by Hoene et. al. [9]. We have used PESQ as a reference for evolutionary modeling. The results of proposed models show a high correlation with PESQ. Moreover, our models are suitable for real-time and non-intrusive estimation of VoIP quality.

The rest of the paper is organized as follows: section 2 talks about the VoIP architecture briefly. To gather the relevant data characterising the speech traffic we have employed a VoIP simulation as described in section 3. Section 4 elucidates how this data is used to evolve the speech quality estimation models. Section 5 presents the results and carries out an analysis of the current research. The paper concludes in section 6 outlining the major achievements and future ambitions.

2 VoIP

As opposed to traditional circuit switched telephony (PSTN), in VoIP the routing of voice conversations takes place over the Internet or an IP based network in the form of packets. The issues related to VoIP communication are governed by

various signaling and transport protocols. Once digitized, human speech is compressed by using a suitable encoding algorithm such as G.711, G.723.1, G.729 and AMR etc. Various speech codecs (encoder/decoder) differ from each other in terms of features such as encoding bit-rate (kbps), algorithmic delay (ms), complexity and, subsequently, speech quality (MOS). After compression and encoding into a suitable format the speech frames are packetized. RTP, UDP and IP packet headers are appended to the frames and the packets are sent to the receiver. During transmission some packets may be lost due to congestion and/or (wireless) transmission errors. The receiver processes the packets and presents them to the playout (dejittering) buffer which is a temporary storage that aims to accumulate enough packets so that they can be played out to the listener as a steady stream as opposed to fragmented clips of voice. The playout buffer seeks to smooth out the variation of the inter-arrival delay (jitter) between the successive voice packets. If packets arrive too late to be played out on time, they are regarded as lost. Consequently, the losses as observed by the application are a superposition of losses due to late arrivals on the losses that occur elsewhere in the VoIP network. After the playout buffer the speech frames are decoded and in doing so any lost frames may be camouflaged by the decoder using a packet loss concealment (PLC) algorithm. Finally the decoded signal is translated in to its acoustic representation. Fig. 1 shows the steps required for mouth-to-ear transportation of voice over an IP network. Silence suppression or discontinuous transmission (DTX) is also supported by VoIP whereby the periods of a conversation when the speaker is silent are not coded or transmitted. DTX is aimed at bandwidth saving. A voice activity detector (VAD) is used to implement DTX.

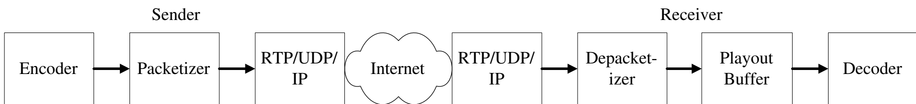


Fig. 1. VoIP system

3 VoIP Traffic Simulation

A simulation based approach was pursued for this research. Such an approach has been employed by various authors such as [10][11]. The main advantage of this approach is that various network distortion scenarios can be emulated precisely. Moreover, the tests are easily repeatable. This section describes the VoIP simulation methodology employed in this work. Before proceeding to the details of actual VoIP simulation environment it is pertinent to discuss the nature of VoIP packet loss. This is described in the following section along with a suitable packet loss model.

3.1 Packet Loss Model

VoIP packet loss is bursty in nature as it exhibits temporal dependency. In the current context the term burst is used to describe the event of a consecutive

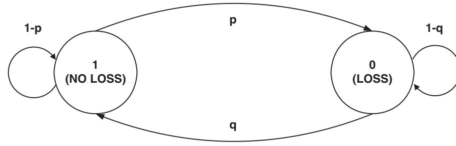


Fig. 2. The Gilbert Model

loss of a number of packets. So, if packet n is lost then normally there is a higher probability that packet $n + 1$ will also be lost. ITU-T Recommendation G.1050 [12], which presents a network model for evaluating multimedia transmission performance over IP, has proposed to use the Gilbert model to capture temporal dependency. Fig. 2 shows the state diagram of this 2-state Markov model.

In this stochastic automaton, p is the conditional probability that the packet numbered $n + 1$ is lost given that packet n is successfully received and q is the converse. $1 - q$ corresponds to the conditional loss probability (clp). Usually $p < 1 - q$. Moreover, the Gilbert model reduces to a Bernoulli model if $p + q = 1$. In (1) mlr corresponds to the mean loss rate and mbl corresponds to the mean burst length.

$$mlr = \frac{p}{p+q}, mbl = \frac{1}{q} \tag{1}$$

The values of p and q can be calculated using the loss length distribution statistics of a network traffic trace.

3.2 VoIP Simulation Environment

This section describes the network simulation environment and the testbed used in this study. A schematic of the simulation environment is shown in Fig. 3. The system includes a speech database, encoder(s)/decoder(s), a packet loss simulator, a speech quality estimation module (PESQ), a parameter extraction module for computing the values of different parameters and a GP based speech quality estimation model. Three popular codecs were chosen in the current research, namely;

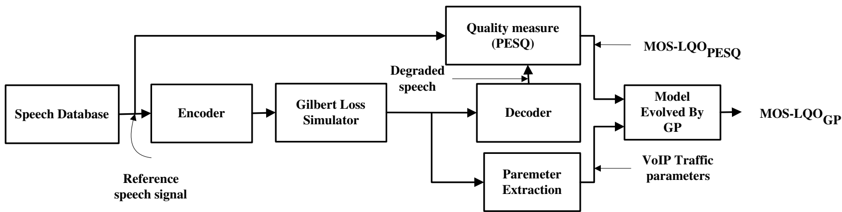


Fig. 3. Simulation system for speech quality estimation model

G.729 CS-ACELP (8 kbps) [13], AMR-NB [14] and G.723.1 MP-MLQ/ACELP (6.3/5.3 kbps) [15]. All of these are based on Linear Predictive Coding of speech. LPC is a scheme whereby the spectral envelop of human speech can be represented in a compressed form. AMR was used in its 7.4 and 12.2 kbps modes whereas G.723.1 was used in its 6.3 kbps mode only. All of these low-bitrate codecs aim at VoIP traffic bandwidth saving. These codecs also have built-in VAD and PLC mechanisms.

The choice of network simulation characteristics was driven by ITU-T Recommendation G.1050 [12] which describes a model for evaluating multimedia transmission performance over an IP network. Bursty packet loss was simulated using the Gilbert model (Fig. 2). It also models packets discarded by the playout buffer due to late arrivals. Packet loss was simulated for different values of *mlr* and *clp*. Twelve different values for *mlr* were chosen; 0, 2.5, 5, 7.5, 10, 12.5, 15, 20, 25, 30, 35 and 40%. The peak loss-rate (i.e. 40%) was kept an order of magnitude higher than that specified for an unmanaged network in ITU-T G.1050 (i.e. 20%) so as to gather more representative data for model derivation. For each value of *mlr*, *clp* was set to 10, 50, 60, 70, and 80%.

After subjecting the VoIP streams under test to various network impairments, they were evaluated using the PESQ algorithm. The PESQ algorithm compares a degraded speech signal with a reference (clean) speech signal and computes an objective MOS score ranging between -0.5 and 4.5, albeit for most cases the output will be between 1.0 and 4.5. It must be mentioned that PESQ simulates a listening test and is optimized to represent the average evaluation (MOS) of all listeners. It is statistically proven that the best possible result that can be obtained from a listening only test is never 5.0, hence it was set to 4.5[1]. The PESQ algorithm is widely acclaimed for its high correlation with the results of formal subjective tests for a wide range of network distortion conditions. It is the current *de jure* standard for objective speech evaluation. In the current context, the MOS scores obtained by the PESQ algorithm and the MOS predicted by the GP based model are differentiated by the abbreviations $MOS\text{-}LQO_{PESQ}$ and $MOS\text{-}LQO_{GP}$ respectively. The term *MOS-LQO* is an acronym for *Mean Opinion Score-Listening Quality Objective* and the various subscripts are used to identify the objective quality estimation models used. This terminology is based on [16].

Altogether, five VoIP traffic parameters have been chosen in the current analysis which form the input variables for evolutionary modeling. These parameters are: codec bit-rate (kbps), packetization interval (PI), frame duration (ms), mlr_{VAD} and mbl_{VAD}. A lower value of bit-rate corresponds to a higher compression of the speech signal, thus resulting in a lower bandwidth requirement at the expense of quality. The packetization interval, which specifies the acoustic information worth a certain duration of time to be contained in a VoIP packet, was varied between 10 to 60 ms. Considerable bandwidth saving can be achieved by encapsulating multiple speech frames in one VoIP packet thus reducing the need for RTP/UDP/IP headers that would have been required for encapsulation and

[1] http://www.opticom.de/technology/pesq.html

transportation of speech frames if they were to be sent individually. However, higher packetization intervals have certain associated drawbacks too. First, the end-to-end delay of the VoIP stream is increased as the sender has to buffer speech frames for a considerable duration before subsequent frames become available by the encoder. Second, for a large packetization interval, typically higher than 40 ms, loss of a single packet results in noticeable degradation of speech quality. Hence, packetization interval presents a trade-off between the speech quality and bandwidth saving. Frame duration has a similar effect on the quality as that of packetization interval. Higher frame durations may have other disadvantages, for instance, in LPC speech signal is assumed to be stationary (non-transient) for a given frame duration. However, for higher frame durations this assumption may considerably deviate from the reality. Thus, a codec with such a feature may obfuscate the final speech content. The parameter extraction module (Fig. 3) is used to obtain the values of the aforementioned parameters from the VoIP traffic stream under test. The corresponding $MOS\text{-}LQO_{PESQ}$ of the decoded VoIP stream under test subjected to these network conditions forms the target output value for training purposes. In actual VoIP applications this information would be gathered by parsing the RTP headers and bitstreams of the encoded frames. The information would then be used as an input for the GP based model to estimate $MOS\text{-}LQO_{GP}$ after processing.

Table 1. Common GP Parameters among all simulations

Parameter	**Value**
Initial Population Size	300
Initial Tree Depth	6
Selection	Lexicographic Parsimony Pressure Tournament
Tournament Size	2
Genetic Operators	Crossover and Subtree Mutation
Operators Probability Type	Adaptive
Initial Operator probabilities	0.5 each
Survival	Half Elitism
Generation Gap	1
Function Set	plus, minus, multiply, divide,sin, cos, log_2, log_{10}, log_e, sqrt, power,
Terminal Set	Random real-valued numbers between 0.0 and 1.0. Integers (2-10). mlr_{VAD}, mbl_{VAD}, PI, br, fd

4 Experimental Setup

As discussed earlier GP was the machine learning algorithm of choice for deriving a mapping between network traffic parameters and VoIP quality. GPLab was used as the preferred GP environment in this study. GPLab is a Matlab toolbox developed by Sara Silva [2]. A total of 4 GP simulations were conducted.

[2] http://gplab.sourceforge.net/

The common parameters of all the simulations are listed in Table 1. In all of the simulations the population size was set to 300. Each simulation was composed of 50 runs whereas each run spanned 50 generations. Adaptive genetic operator probabilities were used [3] [18]. Tournament selection with Lexicographic Parsimony Pressure (LPP) [19] was used in all of the simulations. Survival was based on elitism. The elitist criterion was such that half of the population of a new generation would be composed of best individuals from both parents and children. The other half of the population would be formed of remaining children on the basis of fitness. This elitism criteria is termed as *half elitism* in GPLab.

In simulation 1 mean squared error (mse) was used as the fitness function and tournament size was set to 2. For simulation 2 (and subsequent simulations) scaled mean squared error (MSE_s) was used as the fitness criterion and is given by equation (2).

$$MSE_s(y,t) = 1/n \sum_{i}^{n} \left(t_i - (a + by_i)\right)^2 \qquad (2)$$

where y is a function of the input parameters (a mathematical expression), y_i represents the value produced by a GP individual and t_i represents the target value which is produced by the PESQ algorithm. a and b adjust the slope and y-intercept of the evolved expression to minimize the squared error. They are computed according to equation (3).

$$a = \overline{t} - b\overline{y}, b = \frac{cov(t,y)}{var(y)} \qquad (3)$$

where $\overline{t}$ and $\overline{y}$ represent the mean values of the corresponding entities whereas *var* and *cov* mean the variance and covariance respectively. This approach is known as *linear scaling* and is found to be very beneficial for the symbolic regression tasks with GP [20]. In simulation 2 (and subsequent simulations) *protected* functions were not used. Instead any inputs were admissible to all the functions. For the input values outside the domain of the functions *log, sqrt, division and pow*, NaN (undefined) values are generated. This results in the individual concerned being assigned the worst possible fitness.

The selection criterion in simulations 3 and 4 was based on the notion that population diversity can be enhanced if mating takes place between two, fitness-wise, dissimilar individuals, as suggested by Gustafson et. al. [21]. This selection scheme has been shown to perform better in the symbolic regression domain and, hence, it was employed in this research. This simple addition to the selection criterion only requires one to ensure that mating does not take place between individuals of equal fitness. In simulation 4 the maximum tree depth was changed from 17 to 7 to see if parsimonious individuals with performance comparable to those of earlier simulations can be obtained. Statistics pertaining to simulations and the results are presented in the next section.

[3] Adaptive operator probabilities are discussed on page 31 of the GPLab manual.

5 Results and Analysis

Nortel Networks speech database containing high quality voice signals was used for analysis. The database contains 240 speech files corresponding to two male (m_1, m_2) and two female (f_1, f_2) speakers. Duration of speech signals in the files was between 10-12s. A total of 3360 speech files were prepared for various combinations of afore-mentioned values of network traffic parameters. The simulation parameters include frame duration, bit-rate, packetization interval, *mlr* and *clp*. 70% and 30% of the data of distorted speech files corresponding to speakers m_1 and f_1 were used for training and testing of the evolutionary models respectively. Distorted speech files corresponding to speakers m_2 and f_2 were used to validate the performance of the chosen model against speaker independent data. In other words network traffic parameters and corresponding $MOS\text{-}LQO_{PESQ}$ of 1177, 503 and 1680 speech files were used for training, testing and validation respectively.

Table 2(a) lists the statistics about the MSE of the training/testing data and of final tree size (in terms of number of nodes) of the 4 simulations under consideration. A Mann-Whitney-Wilcoxon test was also performed to decide if a significant difference exists between the simulations. Its results are tabulated in Table 2(b). At 5% significance level a '0' in the tableau indicates that no significant difference exists between the two simulations with respect to that *metric* (i.e. MSE_{tr}, MSE_{te} or $Size$). A '1' indicates the converse and an 'x' marks that the metric is not to be compared with itself.

A keen look at the tables 2(a) and 2(b) shows that simulation 2 (which used linear scaling) performed significantly better than simulation 1. When we compare it with simulation 3, we see that simulation 3 produces significantly smaller trees than simulation 2, albeit with marginally inferior fitness. Finally, simulation 4 exhibits similar traits, as its fitness is marginally worse again, although its trees are significantly smaller. The objective in the current research

Table 2. Statistical analysis of the GP simulations

(a) MSE Statistics for Best Individuals of 50 Runs for Simulations 1-4

	$Sim1$			$Sim2$			$Sim3$			$Sim4$		
Stats	MSE_{tr}	MSE_{te}	$Size$	MSE_{tr}	MSE_{te}	$Size$	MSE_{tr}	MSE_{te}	$Size$	MSE_{tr}	MSE_{te}	$Size$
Mean	0.0980	0.1083	42.6	0.0414	0.0430	38.8	0.0434	0.2788	28.5	0.0436	0.0436	18.0
Std. Dev.	0.0409	0.0507	24.1	0.0040	0.0044	21.2	0.0042	1.0986	15.1	0.0037	0.0060	7.1
Max.	0.2135	0.2656	103	0.0543	0.0568	104	0.0519	6.8911	74	0.0520	0.0782	38
Min.	0.0449	0.0464	8	0.0368	0.0370	5	0.0378	0.0390	9	0.0370	0.0387	8

(b) Results of Mann-Whitney-Wilcoxon Significance Test

	$Sim1$			$Sim2$			$Sim3$			$Sim4$		
Stats	MSE_{tr}	MSE_{te}	$Size$	MSE_{tr}	MSE_{te}	$Size$	MSE_{tr}	MSE_{te}	$Size$	MSE_{tr}	MSE_{te}	$Size$
Sim1	x	x	x	1	1	0	1	1	1	1	1	1
Sim2	1	1	0	x	x	x	1	0	1	1	1	1
Sim3	1	1	1	1	0	1	x	x	x	0	0	1
Sim4	1	1	1	1	1	1	0	0	1	x	x	x

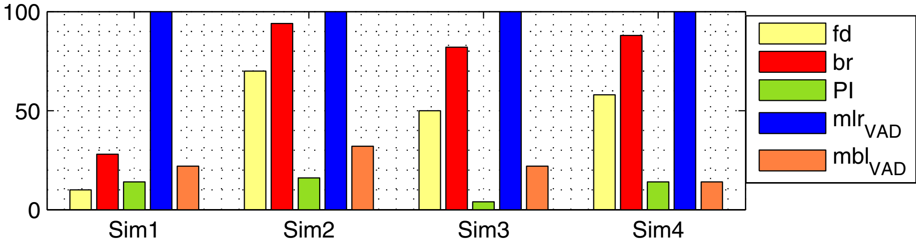


Fig. 4. Percentage of the best individuals employing various input parameters in the 50 runs of each of the four simulations

was to find fitter individuals with small sizes. Hence, simulation 4 was scavenged for plausible solutions.

Fig. 4 delineates the significance of various network traffic parameters in terms of the number of best individuals using them in each of the four GP simulations. It turns out that mlr_{VAD} had a 100% utility in all of the simulations. Codec bit-rate (br) and frame duration (fd) were the second and third most frequently availed parameters respectively. Where as, both PI and mbl_{VAD} have shown advantage in least number of runs of all simulations.

Two of the models derived from this work are shown in this paper by equations (4) and (5). The MSE_s and Pearson's product moment correlation coefficients (σ) of equations (4) and (5) are compared with each other in Table 3. Equation (5) is a function of mlr_{VAD} solely. Whereas equation (4), which was the best model discovered, additionally has br and fd as independent variables. This was the best model of all the runs too. Fig. 5 shows the scatter plots of equation (4) for training and testing data. It is noticeable that equation (5) which is a function of mlr_{VAD} only, however, has comparable fitness to equation (4). Evaluating a single variable would be computationally cheap for a real time analysis. In the light of this and the earlier discussion on Fig. 4 mlr_{VAD} seems to be the most crucial parameter for VoIP quality estimation.

$$MOS - LQO_{GP} = -2.46 \times log\left(cos(log(br)) + mlr_{VAD} \times (br + fd/10)\right) + 3.17 \quad (4)$$

$$MOS - LQO_{GP} = -2.99 \times cos\left(0.91 \times \sqrt{sin(mlr_{VAD})} + mlr_{VAD} + 8\right) + 4.20 \quad (5)$$

As stated earlier ITU-T P.563 is the new recommendation for non-intrusive speech quality estimation. A correlation analysis was done between $MOS\text{-}LQO_{PESQ}$ and the corresponding objective MOS values obtained by ITU-T P.563 ($MOS\text{-}LQO_{P.563}$). It turned out that the correlation coefficients (σ) varied between 0.65-0.82 under various network traffic conditions. This also highlights the significance of current research. It is reiterated to emphasize that ITU-T P.563 is a non-real-time process as it relies upon complex *digital signal processing* techniques to estimate the quality of the speech signal under test. The proposed models, on the other hand, are the functions of network traffic parameters that can be gathered efficiently by parsing VoIP packets.

Table 3. Performance Statistics of the Proposed Models

Data	Equation (4)		Equation(5)	
	MSE_s	σ	MSE_s	σ
Training	0.0370	0.9634	0.0520	0.9481
Testing	0.0387	0.9646	0.0541	0.9501
Validation	0.0382	0.9688	0.0541	0.9531

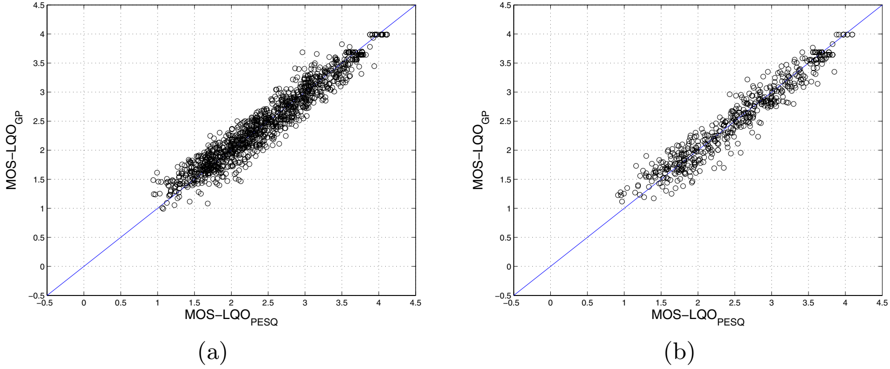


Fig. 5. MOS-LQO predicted by the proposed individual vs MOS-LQO measured by PESQ for (a) training data and (b) testing data for equation 4

6 Conclusions and Future Work

The problem of real-time quality estimation of VoIP is of significant interest. This paper has shown an approach for solving this problem by employing GP. One of the main objectives of this research was to estimate the effect of burstiness on speech quality. It turned out that burst length was least used by the best individuals of various runs. This is due to the fact that the PESQ algorithm does not model the effect of burstiness on speech quality [10] [11]. Hence, the effect of burstiness can be mapped only by conducting suitably designed formal subjective tests [22]. Despite this limitation, PESQ is the best objective quality estimation model and has been used to model the effect of packet loss by various studies. The proposed models are good approximations to PESQ and computationally more efficient. Hence, they are useful for real-time call quality evaluation. For the codecs considered in this study, we have also proposed a model (equation(5)) that is a function of mlr_{VAD} only with performance comparable to the other models. This is considerable since such a model can be deployed conveniently on a wide variety of platforms.

Our results are better than the past research both in terms of performance and nature of the proposed models. For instance, Sun and Ifeachor [6] and Mohamed et. al. [7] [8] proposed ANN based models for VoIP quality estimation

with the number of input parameters ranging between 4–5. However, a major limitation of ANNs is that the model interpretation remains an insurmountable proposition upon successful learning and, as a consequence, there is no direct method for estimating the significance of various input parameters. As stated earlier, evolutionary search prunes off the less significant input parameters leading to simpler models proposed in this paper. Similarly, in their award winning paper Hoene et. al. [9] present a look-up table based VoIP quality estimation model. The various MOS and corresponding parameter values would be stored in a lookup table. In the case the table does not contain a particular value of a parameter, linear interpolation is used to calculate MOS. Moreover, the model is not developed against a wider variety of input parameters. Although codec type is suggested as a network traffic variable in the abstract presentation of their VoIP quality estimation model, the number of codecs is actually restricted to 1 (i.e. AMR codec) in the model proposed therein. Our proposed models are free from such limitations. They can be used to assay the VoIP quality for any values of the input parameters which fall under the permissible range. Moreover, our models have been evolved against highly varying network conditions.

The focus of the current research has been on estimating the effect of those VoIP traffic parameters that affect the listening quality of a telephone call. A future objective would be to derive a model for conversational quality estimation of a call. Conversational quality suffers due to increase in the end-to-end delay of a call. Clearly, our next objective would be to estimate the combined effect of VoIP traffic parameters including the end-to-end delay on call quality.

References

1. ITU-T: Methods for subjective determination of transmission quality. International Telecommunications Union, Geneva, Switzerland. (1996) ITU-T Recommendation P.800.
2. ITU-T: Perceptual evaluation of speech quality (PESQ), an objective method for end-to-end speech quality assessment of narrowband telephone networks and speech codecs. International Telecommunications Union, Geneva, Switzerland. (2001) ITU-T Recommendation P.862.
3. ITU-T: Single-ended method for objective speech quality assessment in narrowband telephony applications. International Telecommunications Union, Geneva, Switzerland. (2005) ITU-T Recommendation P.563.
4. ITU-T: The E-Model, a computational model for use in transmission planning. International Telecommunications Union, Geneva, Switzerland. (1998) ITU-T Recommendation G.107.
5. ITU-T: Methodology for Derivation of Equipment Impairment Factors From Subjective Listening-Only Tests. International Telecommunications Union, Geneva, Switzerland. (2001) ITU-T Recommendation P.833.
6. Sun, L.F., Ifeachor, E.C.: perceived speech quality prediction for voice over ip-based networks. In: IEEE International Conference on Communications (ICC). Volume 4. (2002) 2573–2577

7. Mohamed, S., Cervantes-Perez, F., Afifi, H.: Integrating networks measurements and speech quality subjective scores for control purposes. In: Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM). (2001) 641–649
8. Mohamed, S., Rubino, G., Varela, M.: A method for quantitative evaluation of audio quality over packet networks and its comparison with existing techniques. In: Measurement of Speech and Audio Quality in Networks (MESAQIN). (2004)
9. Hoene, C., Karl, H., Wolisz, A.: A perceptual quality model for adaptive voip applications. In: In Proc. of International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS), San Jose, California, USA (2004)
10. Pennock, S.: Accuracy of the perceptual evaluation of speech quality (pesq) algorithm. In: Measurement of Speech and Audio Quality in Networks (MESAQIN). (2002)
11. Sun, L.F., Ifeachor, E.C.: Subjective and objective speech quality evaluation under bursty losses. In: Measurement of Speech and Audio Quality in Networks (MESAQIN). (2002)
12. ITU-T: Network model for evaluating multimedia transmission performance over internet protocol. International Telecommunications Union, Geneva, Switzerland. (2005) ITU-T Recommendation G.1050.
13. ITU-T: Coding of Speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear-prediction (CS-ACELP). International Telecommunications Union, Geneva, Switzerland. (1996) ITU-T Recommendation G.729.
14. ETSI EN 301 704 V7.2.1: (Digital cellular telecommunications system; Adaptive Multi-Rate (AMR) speech transcoding)
15. ITU-T: Dual rate speech coder for multimedia communication transmitting at 5.3 and 6.3 kbit/s. International Telecommunications Union, Geneva, Switzerland. (1996) ITU-T Recommendation G.723.1.
16. ITU-T: Mean opinion score (MOS) terminology. International Telecommunications Union, Geneva, Switzerland. (2003) ITU-T Recommendation P.800.1.
17. Chu, W.C.: Speech Coding Algorithms: Foundation and Evolution of Standardized Codecs. John Wiley and Sons Inc (2003)
18. Davis, L.: Adapting operator probabilities in genetic algorithms. In: Proceedings of the Third International Conference on Genetic Algorithms, San Mateo, CA (1989)
19. Luke, S., Panait, L.: Lexicographic parsimony pressure. In et. al., W.B.L., ed.: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York (2002) 829–836
20. Keijzer, M.: Scaled symbolic regression. Genetic Programming and Evolvable Machines **5**(3) (2004) 259–269
21. Gustafson, S., Burke, E.K., Krasnogor, N.: On improving genetic programming for symbolic regression. In: Proceedings of the 2005 IEEE Congress on Evolutionary Computation. (2005)
22. ITU-T: Subjective performance assessment of telephone-band and wideband digital codecs. International Telecommunications Union, Geneva, Switzerland. (1996) ITU-T Recommendation P.830.

Training Binary GP Classifiers Efficiently: A Pareto-coevolutionary Approach

Michal Lemczyk and Malcolm I. Heywood

Faculty of Computer Science, Dalhousie University
6050 University Avenue. Halifax. NS. Canada
{lemczyk,mheywood}@cs.dal.ca

Abstract. The conversion and extension of the Incremental Pareto-Coevolution Archive algorithm (IPCA) into the domain of Genetic Programming classification is presented. In particular, the coevolutionary aspect of the IPCA algorithm is utilized to simultaneously evolve a subset of the training data that provides distinctions between candidate classifiers. Empirical results indicate that such a scheme significantly reduces the computational overhead of fitness evaluation on large binary classification data sets. Moreover, unlike the performance of GP classifiers trained using alternative subset selection algorithms, the proposed Pareto-coevolutionary approach is able to match or better the classification performance of GP trained over all training exemplars. Finally, problem decomposition appears as a natural consequence of assuming a Pareto model for coevolution. In order to make use of this property a voting scheme is used to integrate the results of all classifiers from the Pareto front, post training.

1 Introduction

Binary classification problems within the context of a supervised learning paradigm provide the basis for a wide range of application areas under machine learning. However, in order to provide scalable as well as accurate solutions, it must be possible to train classifiers efficiently. Although Genetic Programming (GP) has the potential to provide classifiers with many desirable properties, the computational overhead in doing so has typically been addressed through hardware related solutions alone [9],[2],[5]. In this work we concentrate on how the training process can be made more efficient by evaluating classifier fitness over some adaptive subset of the total training data. To date, the typical approach has been to utilize an active learning algorithm for this purpose, where the Dynamic Subset Selection (DSS) family represents one widely used approach [3],[8],[11].

In this work, an alternative approach to the problem is presented in which the problem is designed as a competition between two populations, one representing the classifiers, the other the data. Progress has recently been made using Genetic Algorithms based on a Pareto formulation of the competitive coevolutionary approach, albeit within the context of player behaviours in gaming environments. To this end, the proposed approach is based on the Incremental

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 229–240, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Pareto-Coevolution Archive (IPCA) algorithm, where this has been shown to address several potential problems with the competitive coevolutionary paradigm i.e., relativism, focusing, disengagement, and intransitivity [1].

The algorithm reported in this work, hereafter denoted the Pareto-coevolutionary GP Classifier (PGPC) is novel in the fact that it extends a Genetic Algorithm "game-playing" context into the domain of GP classification. Furthermore, pruning is utilized to limit the sizes of the IPCA algorithm archives – the point and learner pareto-fronts – to allow for efficient execution. This differs from the method employed in the follow-up of the IPCA algorithm, the Layered Pareto-Coevolutionary Archive (LAPCA) [4], which relies on storing the top N pareto-layers of the archive, keeping the pareto-front in its entirety. Additionally, PGPC differs from the methods utilized by various Evolutionary Multi-Objective Optimization (EMOO) algorithms, which tend to perform clustering on the pareto-front of solutions using the coordinates of candidate solutions to limit the size of the pareto-front [6], [12]. That is to say, the cooperative coevolutionary case of EMOO is able to maintain limits on the size of the Pareto front through similarity measures applied pairwise to candidate solutions. In a GP environment, the design of a suitable similarity metric is not straightforward as learners take the form of programs. As a consequence, this work investigates pruning heuristics that make use of structure inherent in the interaction between learners and training points. Thus, the learner archive is pruned relative to a performance heuristic defined over the contents of the point archive, and the point archive is pruned relative to a heuristic quantifying class distribution and point similarity.

In addition, the GP context requires an alternative approach from those employed previously when resolving which solution from the pareto-front to apply under post training conditions. Specifically, an EMOO context does not face this problem as a individual is identified from the pareto-front of solutions on the basis of a distance calculation. The solution with minimum distance relative to the unseen test condition represents the optimal response. Conversely, under a GP classification context all individuals from the pareto-front provide a label i.e., only under training conditions are we able to explicitly identify which classifier is optimal through the associated classification error. Thus, instead of a single individual representing the optimal strategy for each exemplar, a voting policy is adopted in which all members of the pareto-front provide labels for each exemplar under post training conditions.

1.1 Approach

The co-evolutionary approach of the IPCA algorithm will allow for the "binding" of the learner and training data point subset evolutions, keeping the point subset relevant to the current set of learners. The pareto-front of learners allows the system to explore the search space along different attractors present in the data, and hopefully provide a diverse set of optimal solutions. In regards to the pareto-front of points, each point is pareto-equivalent to the others in the front, and as such provides a "distinction" between the learners that is not duplicated in the

archive. Therefore the pareto-front of points itself is the subset of training data that provides a learning gradient for the learners [1], [4].

The original IPCA algorithm performed no pruning on the learner and point archives. Empirically, the learner archive (pareto-front) remained small, and the point archive which contained the current set of relevant points in addition to the previously relevant set, grew without bounds [1]. In the case of the proposed PGPC algorithm, experiments showed that both the learner and point pareto-fronts grow dramatically on the training data sets, since it may be that each training point is an underlying objective and provides a distinction between learners. To retain efficiency, the previously relevant points may not be stored, nor can the pareto-fronts in their entirety. A pruning method must be adopted to limit the size of the archives.

Furthermore, in the context of a classification problem, a heuristic is required to define how the pareto-front of learners is consolidated to return one classifier per testing point. Since the pareto-front of learners may be diversified to correctly classify subsets of the data, a method to recognize and utilize any structure inherent in the front must be developed such that the most appropriate classifier responds under unseen data conditions. This is generally not a problem within the context of EMOO as solutions take the form of a point in a coordinate space. Identifying which individual represent the best solution is resolved through the application of a suitable distance metric. Under the GP (classification) domain, solutions take the form of models providing a mapping from a multi-dimensional input space to a 1-dimensional binary output space. Thus, on unseen data it is not possible to associate exemplars to models a priori. This problem is addressed in this work by adopting a simple voting scheme over the contents of the learner archive.

2 The Pareto-coevolutionary GP Classifier Algorithm

In terms of the GP individuals or learners, a tree-structured representation is employed, whereas individuals in the point population(s) index the training data. The classical GP approach for interpreting the numerical GP output (*gpOut*) in terms of a class label is utilized, or

$$\text{IF } (gpOut \leq 0.0) \text{ THEN (return class 0), ELSE (return class 1)} \qquad (1)$$

The PGPC algorithm utilizes four populations of individuals: (1) a fixed size learner population which provides the exploratory aspect of the learner evolution. (2) a learner archive which contains the pareto-front of learners, bound by a maximum size value. (3) a fixed size point population (point population $<<$ training exemplar count). (4) a point archive which describes the current subset of training points relevant to the learner archive, bound by a maximum size value. Figure 1 summarizes the organization of data dependencies for each step of the algorithm.

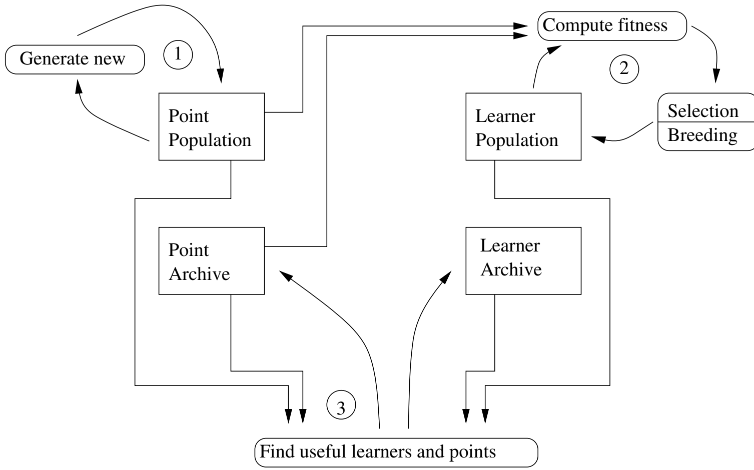


Fig. 1. The PGPC architecture

The PGPC algorithm consists of the following steps performed at each generation of evolution:

Step 1: Generate points in the point population: Since the points are indices within the training data, a crossover operator makes little sense. Therefore, only mutation was utilized to generate the point population, with mutation being performed on each population member. Furthermore, to ensure class balance within the point population, each half of the population is randomly filled with points belonging to the same class, Figure 1 point 1.

Step 2: Generate learners in the learner population: The canonical tree-structured model of GP is assumed [7], although the PGPC algorithm could utilize any standard GP model for the representation and definition of search and selection operators. In this case fitness proportionate selection is assumed, with fitness being calculated over the contents of both the point population and point archive, Figure 1 point 2.

Step 3: The following steps deal with the entry criteria for the point and learner archives, Figure 1 point 3:

3.a: Compute the set of useful points regarding the learner population and archive: As per IPCA; if a newly generated learner is dominated by the learner archive or contains equal values (evaluated over the point archive), and the addition of a new point into the evaluation set provides a distinction such that the generated learner is pareto-equivalent to the archive with no equal values, the point is inserted into the archive.

3.b: Compute the set of useful learners regarding the point population and archive: As per IPCA; any generated learner that is pareto-equivalent to the archive with no equal values (again, evaluated over the point archive) enters the archive. Furthermore, if a generated learner is non-dominated, and a generated point defeats it, they both enter their respective archives.

3.c: Remove duplicates in the learner and point archives and newly-dominated learners in the learner archive: As per IPCA.

To maintain the efficiency of the algorithm, a limit on the archive sizes is enforced. This limit may be thought of as a tunable parameter of efficiency vs accuracy. However the relationship between the two may depend on the classification problem, as the number of underlying objectives which the point archive strives to evolve towards may vary. Moreover, the number of individuals qualifying for inclusion in the point or learner archives is also a dynamic property, with a significant difference between the number qualifying in early versus later generations.

Within the context of the pruning algorithm, the first assertion will be that a newly generated learner or point should enter the archive at the possible cost of evicting an older member. This assertion will help avoid stagnation within the archive at the risk of possible regression or forgetting. Alternate insertion basis may be considered and evaluated in the future. Within this framework, all that remains is to provide a basis for selection of an archive individual for replacement.

For pruning the learner archive, the proposed greedy approach consists of removing the learner with the worst performance against the point archive (with the measure being the number of incorrectly classified instances). Within this view, a learner to be removed may have entered the archive by simply correctly classifying one training archive point while misclassifying the remainder, therefore removing the "worst" learner deletes some of the explorative diversity of the archive in favour of increased average accuracy.

For pruning the point archive, the proposed basis utilizes the genotypic information of the point co-ordinates to delete one of the two closest points, distance defined using the Euclidean metric, adhering to the following criteria: the two points must be of the same class, and that class must be over-represented in the point archive. This approach will promote class-balanced diversity in the point archive, while preserving the points which define boundaries between clusters of points.

Finally, in order to attain a measure of classification performance on testing data at the completion of training, the learner pareto-front must be interpreted to provide one class prediction per testing point. To this end an "Average archive value" voting scheme is used in which each pareto-front learner provides one vote for their class prediction of the input testing point. The class with the majority of the votes is selected as the system's prediction for the corresponding data point. Such a scheme is adopted to make use of the aforementioned learner pruning bias in which learners are rewarded for maximizing the number of correctly

classified exemplars. Thus, we expect classification by consensus as opposed to outright specialization, in which case a more sophisticated voting policy might be required.

3 Experiments

Evaluation of the proposed PGPC algorithm will be performed against canonical tree-structured GP and three alternative active learning algorithms. Indeed all methods will share the same canonical GP model. The three active learning comparison algorithms use a limit on the number of training exemplars to provide a more accurate comparison with the PGPC classifier. These algorithms will allow for an evaluation of the IPCA-based dynamics and solution space search efficiency. They differ from the canonical GP algorithm only in the active learning algorithm used to identify the subset of training exemplars over which fitness evaluation takes place. Section 3.2 summarizes the properties of each alternative algorithm.

Post training evaluation of classification performance is conducted using a single classification “score”. This is based on a combined equal weighting of detection and false positive rate, or

$$Score = \frac{\frac{TruePositives}{Positives} + (1 - \frac{FalsePositives}{Negatives})}{2} \tag{2}$$

Adoption of such a metric will establish how robust alternative schemes are to any under representation of the minor class. Algorithm efficiency is measured in terms of run-time on a common machine under the same conditions.

The classification data sets used in the experiments consist of: Adult, and KDD99[1]. Each data set is considered a two-class problem, with the training partition for Adult being set at 75%. The Adult data set consists of 33916 training points, and 11306 testing points (i.e., exemplars with missing features are not included); each having a dimension of 14 features. The KDD99 set consists of 494020 and 311027 training and testing points, respectively; with each point having a dimension of 41 features. In order to cast the problem as a binary classification problem we concentrate on separating the class representing ‘normal’ from the other four classes. Both data sets are unbalanced with approximately 20 percent in-class exemplars in KDD99 and 15 percent in-class exemplars in Adult.

The relevant hardware of the test machine utilized for the run-time experiments consists of: Pentium 4, 2.60GHz HT, 800MHz FSB. 1GB DDR400 RAM. 36GB SATA 10K-RPM Hard Drive. The GP implementation common to all five methods benchmarked was based on the lilGP[2] framework, running on Fedora Core 3 Linux.

[1] Available at:
http://www.ics.uci.edu/~mlearn/MLSummary.html [Adult]
http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html [KDD99]

[2] http://garage.cps.msu.edu/software/lil-gp

3.1 Parameters

The tree-structured GP parameters common to all of the algorithms are summarized in Table 1. The sizes of the populations and archives of the PGPC algorithm will all be set to a common value of 25.

Although the learner archive is the set of learners constituting an "answer", the learner population is still used for exploration and the evolution of the learners. Therefore the sum of both the archive and population sizes is used to provide an equivalent limit on the number of learners utilized by the comparison algorithms, Table 2. The same holds true for the point population and archive, since they are both used in the evaluation of the fitness of a learner, they constitute the number of points that the algorithm can "access". Therefore the comparison subset selection algorithms utilize a subset size set to be the sum of the two, Table 2.

Due to the stochastic nature of the GP based algorithms, performance is reported over a total of 30 different population initializations per experiment.

Table 1. GP parameters common to all of the algorithms

Parameter	Value
Number of generations	500
Individual initialization method	Half and half
Individual initialization depth	2-6
Individual maximum nodes	1000
Individual maximum depth	17
Learner breeding phase selection method	Fitness Proportionate
Learner breeding phase operators	Crossover, Mutation
Learner breeding phase operator frequencies	0.8, 0.2
Function Set	*, /, +, -, sin, cos, exp, sqrt

Table 2. Population and Point Subset sizes for comparison algorithms

Model	Regular	Cycling	Random	DSS
Learner Population Size	50	50	50	50
Point Subset Size	as per original Training Set	50	50	50

3.2 Comparison Algorithms

Canonical tree-structured GP: The base line comparison algorithm takes the form of a canonical tree-structured GP classifier [7] (denoted as "Regular"), consisting of only one learner population. At every generation, the fitness of each learner is computed using the entire training data set. The absolute switching function wrapper maps the *gpOut* value of the individual against the training

data point class, equation (1), and the number of correct mappings (classifications) is recorded and normalized into a fitness value (accuracy). The fitness values of the individuals are used to perform fitness proportionate selection for breeding the next generation of individuals. Upon completion of the evolution, the fittest individual is used to classify the testing data using the same switching function[3].

Dynamic subset selection (DSS): This comparison algorithm is an implementation of the DSS algorithm [3]. Each training exemplar has an associated age and difficulty value; corresponding to the number of generations elapsed since that exemplar was last utilized in the subset, and the number of correct classifications of it when it was utilized. At each generation, a weighing of the two values is performed to yield a selection probability, or

$$PointWeight_p = Difficulty_p^{1.0} + Age_p^{3.5} \tag{3}$$

The selected points define a subset over which learners are evaluated in the current generation.

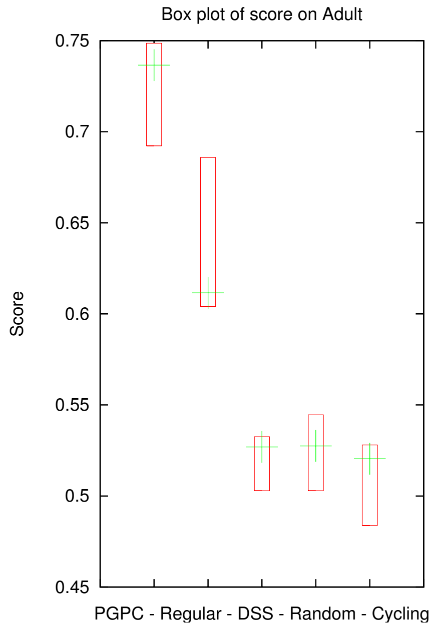
Random subset selection: This algorithm is based on the "Stochastic sampling" method described in [10]. Specifically, individuals are evaluated over a subset of training data points selected randomly with uniform probability and used to compute the fitness of the individual.

Cycling subset selection: At any evaluation of any individual, a global index into the training data is incremented. Training points subsequent to the index are utilized to compute the fitness. Wrap-around is used to resolve the special case associated with the end of the training data set.

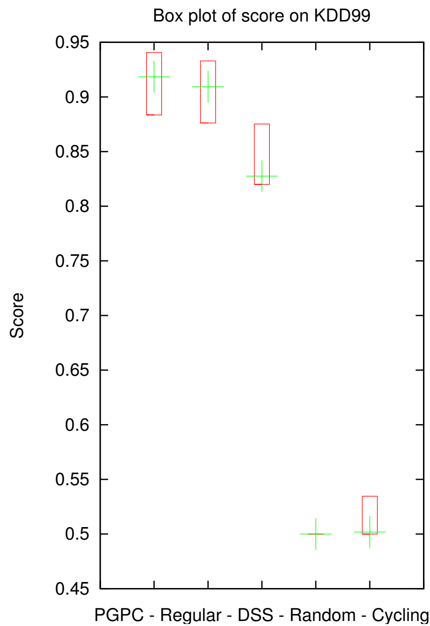
4 Results

Figure 2 illustrates test set performance using first quartile, median, and third quartile under the 'score' metric of equation (2). It is apparent that the PGPC algorithm matched or bettered all of the comparison algorithms, including the Regular GP algorithm. This indicates that the proposed algorithm has indeed provided an effective alternative mechanism for selecting subsets of training exemplars to perform the learner evolution upon. Moreover, although the fitness metric was based on classification count during training, it is also apparent that degenerate solutions were successfully avoided in the ensuing solutions (degenerates are equivalent to a score of 0.5). Conversely, all three alternative subset selection algorithms returned performance scores equivalent to degenerate solutions on the Adult data set. DSS performed better on KDD99, although never as good as PGPC and Regular GP; whereas both random and cycling subset selection performed worse on KDD99 than on the Adult data set. The consistently

[3] The score metric of equation (2) is only employed in the post training evaluation of performance.



(a) Adult



(b) KDD99

Fig. 2. Test data box plot as evaluated using the 'score' metric. Note: the variance of the Random algorithm on KDD99 is minimal, with the difference between the minimum and maximum value being 0.005215 percent, yielding an insignificant box plot.

poor behavior of the Random sampling algorithm on KDD99 was associated with the population being dominated by degenerate solutions that labeled all exemplars as out of class, where this represents around 400,000 of the 500,000 training exemplars.

Table 3. Median scores and run-times in seconds of the various algorithms upon the Adult and KDD99 data sets

Algorithm	Median score (Adult)	Median time (Adult)	Median score (KDD99)	Median time (KDD99)
PGPC	0.736611	41.38	0.918419	56.97
Regular	0.611569	1973.74	0.909291	40347.74
DSS	0.526903	11.29	0.827497	120.87
Random	0.527521	3.63	0.500000	4.20
Cycling	0.520470	3.46	0.501884	2.58

In terms of execution efficiency, Table 3, the PGPC algorithm exhibited a speedup of 48 (Adult) to 708 (KDD99) with respect to Regular GP. The Random and Cycling Subset Selection algorithms were naturally the fastest, but also resulted in degenerate solutions for both data sets. The DSS algorithm was significantly faster than PGPC under Adult, however, was actually slower than PGPC under the larger KDD99 data set by a factor of two.

In summary, the PGPC algorithm was able to match (KDD) or better (Adult) the classification scores of the regular GP algorithm whilst providing a significant speedup. Moreover, as the size of the training data set increased, the computational effectiveness of the PGPC algorithm improves, Table 3. Thus, DSS takes twice as long to provide solutions on the KDD data set than PGPC, for no improvement in classification score. Conversely, the naive schemes for building subsets of exemplars, Random and Cycling, are very fast, but do not provide a useful model for learning, barely reaching a 50 percent classification score i.e., degenerate solutions were the norm.

5 Conclusion

An algorithm employing the coevolution of both classifiers and training data subset members within a Genetic Programming environment was presented. Comparisons were made to a traditional GP classifier employing the training data in its entirety, in addition to classifiers using only a subset of the data selected via either a Random, Cycling, or DSS method.

With regards to classification performance, the PGPC algorithm outperformed each of the comparison algorithms, indicating that the algorithm does not compromise classification performance. We conclude that even with a small subset of training points, the pareto-evolutionary approach to learner

and point co-evolution may generate superior or equivalent classification performance. Moreover, the PGPC algorithm was particularly effective at avoiding degenerate solutions; where this is particularly useful on problems described by large unbalanced data sets.

In the case of training efficiency, the training point subset selection of PGPC provides a dramatic increase in execution speed, without recourse to specialist hardware. Moreover, the approach becomes increasingly effective as the number of exemplars in the data set increases. Finally, we note that as the entire Pareto front of learners takes part in the solution, problem decomposition is supported as an additional side effect of adopting a coevolutionary paradigm.

Future work will evaluate the significance of different point and learner archive pruning schemes, as well as qualify the significance of pruning in practice. The latter is particularly relevant with regards to the impact of 'regression' or 'forgetting' on the behavior of the point and learner archives.

Acknowledgments

The authors gratefully acknowledge the support of CFI New Opportunities, NSERC Discovery, and MITACS grants (Canadian government), and SwissCom Innovations Inc. (Switzerland).

References

1. E. D. de Jong. The incremental pareto-coevolution archive. In *GECCO (1)*, volume 3102 of *Lecture Notes in Computer Science*, pages 525–536. Springer, 2004.
2. G. Folino, C. Pizzuti, and G. Spezzano. Ensemble techniques for parallel genetic programming based classifiers. In *Proceedings of the 6th European Conference on Genetic Programming (EuroGP)*, volume 2610 of *Lecture Notes in Computer Science*, pages 59–69. Springer, 2003.
3. C. Gathercole and P. Ross. Dynamic training subset selection for supervised learning in genetic programming. In *PPSN*, volume 866 of *Lecture Notes in Computer Science*, pages 312–321. Springer, 1994.
4. E. D. Jong. Towards a bounded pareto-coevolution archive. In *Proceedings of the Congress on Evolutionary Computation CEC-04*, volume 2, pages 2341–2348, 2004.
5. H. Juille and J. Pollack. Massively parallel genetic programming. In *Advances in Genetic Programming, 2nd ed.*, pages 339–358. MIT Press, Cambridge, MA, USA, 1996.
6. J. Knowles and D. Corne. The pareto archived evolution strategy: A new baseline algorithm for pareto multiobjective optimization. In *Proceedings of the Congress on Evolutionary Computation*, volume 1, pages 98–105, Mayflower Hotel, Washington D.C., USA, 6-9 1999. IEEE Press.
7. J. R. Koza. *Genetic Programming II.* Cambridge, Mass.:MIT Press, 1994.
8. C. Lasarczyk, P. Dittrich, and W. Banzhaf. Dynamic subset selection based on a fitness case topology. *Evolutionary Computation*, 12(2):223–242, 2004.
9. P. Nordin. A compiling genetic programming system that directly manipulates the machine code. In *Advances in Genetic Programming.*, pages 311–334. MIT Press, Cambridge, MA, USA, 1994.

10. P. Nordin and W. Banzhaf. An on-line method to evolve behavior and to control a miniature robot in real time with genetic programming. *Adaptive Behavior.*, 5(2):107–140, 1996.
11. D. Song, M. Heywood, and A. Zincir-Heywood. Training genetic programming on half a million patterns: An example from anomaly detection. *IEEE Transactions on Evolutionary Computation*, 9(3):225–239, 2005.
12. E. Zitzler and L. Thiele. Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.

A Comprehensive View of Fitness Landscapes with Neutrality and Fitness Clouds

Leonardo Vanneschi[1], Marco Tomassini[2], Philippe Collard[3], Sébastien Vérel[3], Yuri Pirola[1], and Giancarlo Mauri[1]

[1] Dipartimento di Informatica, Sistemistica e Comunicazione (D.I.S.Co.), University of Milan-Bicocca, Milan, Italy
[2] Computer Systems Department, University of Lausanne, Lausanne, Switzerland
[3] I3S Laboratory, University of Nice, Sophia Antipolis, France

Abstract. We define a set of measures that capture some different aspects of neutrality in evolutionary algorithms fitness landscapes from a qualitative point of view. If considered all together, these measures offer a rather complete picture of the characteristics of fitness landscapes bound to neutrality and may be used as broad indicators of problem hardness. We compare the results returned by these measures with the ones of negative slope coefficient, a quantitative measure of problem hardness that has been recently defined and with success rate statistics on a well known genetic programming benchmark: the multiplexer problem. In order to efficaciously study the search space, we use a sampling technique that has recently been introduced and we show its suitability on this problem.

1 Introduction

In the context of Evolutionary Algorithms (EAs), Neutrality of fitness landscapes has been widely studied in the last few years. Nevertheless, many contributions that have appeared use very different concepts and measures between them to express neutrality. For instance, in [11], Reidys and Stadler define the family of additive random landscapes where both neutrality and ruggedness of fitness landscapes can be tuned; in [14], Toussaint and Igel talk of the suitability of the design of neutral encodings to improve the efficiency of EAs; in [3], Collard et al. introduce the concept of synthetic neutrality and study its effects on the evolvability of Genetic Algorithms (GAs); in [22,20,21], Yu and Miller show that increasing the search space's size by artificially introducing neutral neighbors to some individuals, can help Cartesian Genetic Programming (GP) to navigate some restricted fitness landscapes, focusing on the choice of the representation and how it affects the amount of neutral neighborhood in a fitness landscape (these results have been recently criticized by Collins in [5]). If on the one hand this multiplicity of different concepts and formalisms has contributed to fortify the belief that neutrality plays an important role in the search process of EAs from many different points of view, on the other we think that uniformity in treating neutrality is missing and we fear that this may lead to ambiguous and sometimes confusing conclusions. In other words, we strongly agree with Geard [6] that *the way* in which neutrality is defined is crucial in determining its role and that the choice of different neutrality frameworks and formalizations may lead to different, and in some cases even conflicting, conclusions.

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 241–250, 2007.
© Springer-Verlag Berlin Heidelberg 2007

One of the main goals of this paper is establishing a precise set of *neutrality measures*, each of which aimed at formalizing a particular aspect of fitness landscapes bound to neutrality. These measures (some of which already introduced in [16]) are: the *average neutrality ratio*, the *average Δ-fitness*, the *non-improvable* and "*non-worsenable*" *solutions ratio* and the *profitable* and *unprofitable mutations ratio*. Each of them is calculted on *neutral networks*. None of them brings a sufficient amount of information if considered alone, since each of them focus only on a particular feature of the landscape, but the joint analysis of all of them should allow us to have a rather complete picture of fitness landscapes, especially those related to neutrality. Furthermore, even though a bound between neutrality and problem difficulty has often been hypothesized, neutrality has never been presented together with other difficulty measures before, in order to check if the respective results are consistant between each other or not. In this paper, we compare the qualitative results returned by our neutrality measures with the quantitative results returned by the Negative Slope Coefficient (NSC) (a hardness measure that has recently been proposed in [18,15]) and we hope that the results returned by our neutrality measures may support and strengthen the ones of the NSC. For our empirical study, we use two different versions of the multiplexer problem, induced by two different sets of functional operators: {IF} and {NAND}. Finally, as discussed in [15], the shape and features of the boolean functions fitness landscapes make them hard to study by means of uniform random samplings and thus more sophisticated sampling methods are needed. In this paper we use a new, and more elaborate, sampling methodology of the search space and neighborhood that has been first defined in [16].

This paper is structured as follows: in section 2 we introduce some notions that will be used in this paper and we present NSC results for the two chosen instances of the multiplexer problem. Section 3 presents the view of neutrality features of these two landscapes, as offered by our quantitative neutrality measures. Finally, section 4 discusses the results, concludes the paper and offers hints for future research activity.

2 Definitions and Preliminary Results

Fitness Landscapes and Neutrality. Using a landscape metaphor to gain insight about the workings of a complex system originates with the work of Wright on genetics [19]. A simple definition of fitness landscape in EAs is a plot where the points in the horizontal plane represent the different individual genotypes in a search space (placed according to a particular *neighborhood relationship*) and the points in the vertical direction represent the fitness of each one of these individuals [9]. Generally, the neighborhood relationship is defined in terms of the genetic operators used [17,9,15]. This can be done easily for unary genetic operators like mutation, but it is clearly more difficult if binary or multi-parent operators, like crossover, are considered. Formal definitions of fitness landscape have been given (e.g. in [13]). Following these definitions, in this work a fitness landscape is a triple $\mathcal{L} = (\mathcal{S}, \mathcal{V}, f)$ where $\mathcal{S}$ is the set of all possible solutions, $\mathcal{V} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ is a neighborhood function specifying, for each $s \in \mathcal{S}$, the set of its neighbors $\mathcal{V}(s)$, and $f : \mathcal{S} \rightarrow \mathbb{R}$ is the fitness function. Given the set of variation operators, $\mathcal{V}$ can be defined as $\mathcal{V}(s) = \{s' \in \mathcal{S} | s'$ can be obtained from s by a single variation$\}$. In some cases, as for many GP boolean problems, even though the size of the search

space $\mathcal{S}$ is huge, f can only assume a limited set of values. Thus, a large number of different solutions have the same fitness. In this case, we say that the landscape has a high degree of neutrality [11]. Given a solution s, a particular subset of $\mathcal{V}(s)$ can be defined: the one composed by neighbor solutions that are also neutral. Formally, the *neutral neighborhood* of s is the set $\mathcal{N}(s) = \{s' \in \mathcal{V}(s) | f(s') = f(s)\}$. The number of neutral neighbors of s is called the *neutrality degree* of s and the ratio between neutrality degree and cardinality of $\mathcal{V}(s)$ is the *neutrality ratio* of s. Given these definitions, we can imagine a fitness landscape as being composed by a set of (possibly large) *plateaus*. More formally, a *neutral network* [12] can be defined as a graph connected component $(\mathcal{S}, E_{\mathcal{N}})$ where $E_{\mathcal{N}} = \{(s_1, s_2) \in \mathcal{S}^2 | s_2 \in \mathcal{N}(s_1)\}$. Finally, we define the *fitness of a neutral network* (or *network fitness*) as the fitness value shared by all individuals of this neutral network.

Negative Slope Coefficient. Evolvability of a solution related to an operator [1] can be studied by plotting the fitness values of individuals against the fitness values of their neighbours. Such a plot has been presented in [4,2] and called *fitness cloud*. A possible algorithm to generate fitness clouds was proposed in [15]. This algorithm essentially corresponds to the sampling produced by a set of n stochastic hill-climbers at their first iteration after initialisation. The Negative Slope Coefficient (NSC) has been defined to capture with a single number some interesting characteristics of fitness clouds. It can be calculated as follows: the abscissas of a fitness cloud can be partitioned into a certain number of separate bins $\{I_1, I_2, \ldots, I_m\}$. Let $X_1, X_2, \ldots, X_m$ be the averages of the abscissas of the points contained in bins $I_1, I_2, \ldots, I_m$, respectively, and let $Y_1, Y_2, \ldots, Y_m$ be the averages of the ordinates of the points in $I_1, I_2, \ldots, I_m$. The set of points (X_i, Y_i) can be seen as the vertices of a polyline, which effectively represents the "skeleton" of the fitness cloud. For each of the segments of this, we can define a *slope*, $S_i = (Y_{i+1} - Y_i)/(X_{i+1} - X_i)$. Finally, the negative slope coefficient is defined as $\text{NSC} = \sum_{i=1}^{m-1} min\ (0, S_i)$. The hypothesis proposed in [15] is that the NSC should classify problems in the following way: if NSC$=0$, the problem is easy; if NSC<0 the problem is difficult and the value of NSC quantifies this difficulty: the smaller its value, the more difficult the problem. The justification put forward for this hypothesis was that the presence of a segment with negative slope would indicate a bad evolvability for individuals having fitness values contained in that segment as neighbours would be, on average, worse than their parents. Pros and cons of this measure have been discussed in [18,15].

Genetic Operators and Neighborhood. Standard crossover or subtree mutation [8] generate very complex neighborhoods. In this paper, we consider a simplified version of the inflate and deflate mutation operators first introduced in [15,17] (also called structural mutation operators in those works): (1) *Strict deflate mutation*, which transforms a subtree of depth 1 into a randomly selected leaf chosen among its children. (2) *Strict inflate mutation*, which transforms a leaf into a tree of depth 1, rooted in a random operator and whose children are a random list of variables containing also the original leaf. (3) *Point terminal mutation*, that replaces a leaf with another random terminal symbol. This set of genetic operators (already introduced in [16] and called *Strict-Structural*, or *StSt*, mutation operators) is easy enough to study and provides enough exploration power to GP. For instance, *StSt* mutations present two important properties: (i) each

mutation has an inverse: let M be the set of *StSt* mutation operators and let $\mathcal{S}$ be the set of all the possible individuals (search space). For each pair of individuals $(i, j) \in \mathcal{S}$, if an operator $m \in M$ exists such that $m(i) = j$, then an operator $m^{-1} \in M$ such that $m^{-1}(j) = i$ always exists; (ii) for each pair of solutions $(i, j) \in \mathcal{S}$, a sequence of mutations which transforms i into j exists. See [16] for the formal proofs of these properties. Thus, the associated graph $(\mathcal{S}, \mathcal{V})$ of fitness landscape is undirected (given the (i) property) and connected (given the (ii) property) graph.

The Multiplexer Problem. The goal of the k-multiplexer [8] problem is to design a boolean function with k inputs and one output. The first x of the k inputs can be considered as address lines. They describe the binary representation of an integer number. This integer chooses one of the 2^x $(= k - x)$ remaining inputs. The correct output for the multiplexer is the input on the line specified by the address lines. The terminals are the k variable inputs to the function. The fitness function of a GP individual E is calculated as the number of input data for which E does not return the same value as the target function. In this paper, the fitness values have always been normalized into the $[0, 1]$ range, by dividing them by 2^k, where k is the problem's order. Thus, from now on a solution with fitness equal to 0 represents an optimal solution, while 1 is the worst possible fitness value. In this paper, we have used two different sets of non-terminals: $\{\text{IF}\}$ (where $\text{IF}(x, y, z)$ is a ternary boolean function which returns y if x is *true* and z otherwise) and $\{\text{NAND}\}$. We have chosen these two sets because they are small enough to limit the cardinality of the search space but rich enough to represent some perfect solutions. These two sets of boolean operators induce two landscapes (indicated by $\mathcal{L}_{(k,h)}^{\{\text{IF}\}}$ and $\mathcal{L}_{(k,h)}^{\{\text{NAND}\}}$ from now on, where k is the problem order and h is the predetermined tree depth limit) the first of which is generally easy for GP, while the second is hard. This fact is confirmed by the experimental results shown in table 1, where the values of the success rate (*SR*) for three different mutation rates and NSC are reported for both landscapes. The success rate results have been obtained by executing 100 indepen-

Table 1. Values of the success rate for three different mutation rates and of the NSC for the 6-multiplexer problem using two different sets of operators to build the individuals. The fitness landscapes induced by these two sets of operators clearly have different difficulties for GP.

Set Of Operators	$SR(p_m = 0.95)$	$SR_{(p_m=0.5)}$	$SR_{(p_m=0.25)}$	NSC
$\{\text{IF}\}$	1	0.98	0.71	0
$\{\text{NAND}\}$	0	0	0	-0.21

dent GP runs using the 6-multiplexer problem, maximum tree depth for the individuals equal to 6 for the landscape induced by $\{\text{NAND}\}$ and to 5 for the landscape induced by $\{\text{IF}\}$ (the choice of these values for the tree depths are motivated later), population of size 100, ramped half-and-half population initialization, tournament selection of size 10, *StSt* mutations as genetic operators. Only one *StSt* mutation operator has been applied with a certain probability p_m. 100 GP runs have been executed with $p_m = 0.95$ (column 2 of table 1), 100 separate runs have been executed with $p_m = 0.5$ (column 3)

and 100 further runs have been executed with $p_m = 0.25$ (column 4). The choice of the particular mutation operator has been done each time uniformly at random between the three *StSt* mutations. A run has been considered successful when an individual with a lower fitness than 0.15 has been found. The results related to the NSC reported in table 1 (column 5) have been obtained by generating a sample of 40000 individuals with the Metropolis-Hastings algorithm and, for each of them, a neighbor by applying one *StSt* mutation. Once again, the choice of the particular mutation operator to generate each neighbor has been done uniformly at random between the three *StSt* mutations.

Sampling Methodology. In [9] uniform random samplings have been used for studying boolean function landscapes. In [15] importance sampling techniques such as Metropolis and Metropolis-Hastings [10] have been proposed. Even though the resuls obtained were satisfactory for the purposes of those works, still those samples did not capture some important characteristics of the fitness landscape (see [16] for a detailed discussion). In this paper, we use a methodology aimed at generating samples containing trees of many (possibly all) different fitness values and forming connected neutral networks, if possible. This technique is composed by three steps: *modified Metropolis*, *vertical expansion* and *horizontal expansion*. Modified Metropolis generates a sample S of individuals with as many different fitness values as possible. The vertical expansion tries to enrich S by adding to it some *non-neutral* neighbors of its individuals. Finally, the horizontal expansion tries to enrich S by adding to it some *neutral* neighbors of its individuals. This methodology has been presented in [16] and it is not described here to save space.

3 Neutrality Results

In this section we present a study of neutrality of $\mathcal{L}^{\{\text{NAND}\}}_{(6,6)}$ and $\mathcal{L}^{\{\text{IF}\}}_{(6,5)}$, which are the largest search spaces respectively induced by $\{\text{NAND}\}$ and $\{\text{IF}\}$ that we have been able to study with our computational resources. The difference in the tree depth limit for the two landscapes is due to the fact that NAND is an operator with arity 2 while IF is an operator of arity 3. Thus, given a fixed tree depth, the trees that can be built with IF are on average larger than the ones that can be built with NAND. Figure 1 shows the fitness distributions (that is the frequence of fitness value) of the samples of $\mathcal{L}^{\{\text{NAND}\}}_{(6,6)}$ and $\mathcal{L}^{\{\text{IF}\}}_{(6,5)}$ that we have generated with Metropolis-Hasting sampling technique. For $\mathcal{L}^{\{\text{IF}\}}_{(6,5)}$, all the sampled fitness values are included into the range $[0, 0.7]$; in other words, no *bad* individual has been sampled. This is probably a characteristic of the complete search space (and it is not due to a bias of our samplig technique); in fact, we have exhaustively generated all the possible individuals of $\mathcal{L}^{\{\text{IF}\}}_{(3,2)}$ and we have observed that no tree with fitness larger than 0.7 exists also in that (similar, although much smaller) search space. On the other hand, for $\mathcal{L}^{\{\text{NAND}\}}_{(6,6)}$ large part of the sampled individuals have a bad fitness value (included into the range $[0.75, 1]$). Also this characteristic is analogous to what happens in the similar, but smaller, search space $\mathcal{L}^{\{\text{NAND}\}}_{(3,3)}$ that we have been able to exhaustively generate, where the largest number of individuals had fitness equal to 0.75 and the majority of the individuals had bad fitness values. Finally, we point out

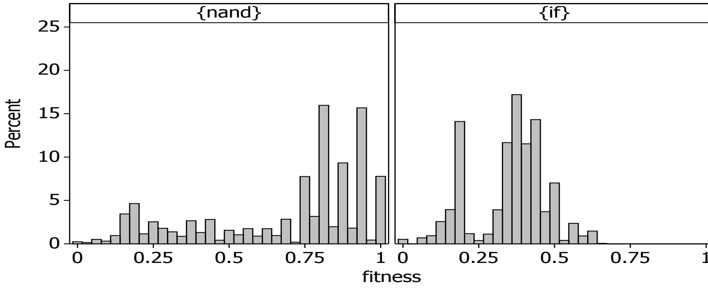


Fig. 1. Fitness distribution of $\mathcal{L}^{\{\mathrm{NAND}\}}_{(6,6)}$ (left part) and $\mathcal{L}^{\{\mathrm{IF}\}}_{(6,5)}$ (right part)

that with our sampling technique we have been able to generate individuals with many different fitness values, which is quite unusual for boolean landscapes (as pointed out, for instance, in [15]). Figure 2 reports the average neutrality ratio of neutral networks scatterplots for $\mathcal{L}^{\{\mathrm{NAND}\}}_{(6,6)}$ (left part) and $\mathcal{L}^{\{\mathrm{IF}\}}_{(6,5)}$ (right part) as a function of fitness value. The average neutrality ratio, $\bar{r}$ is defined as the mean of the neutrality ratios (as defined

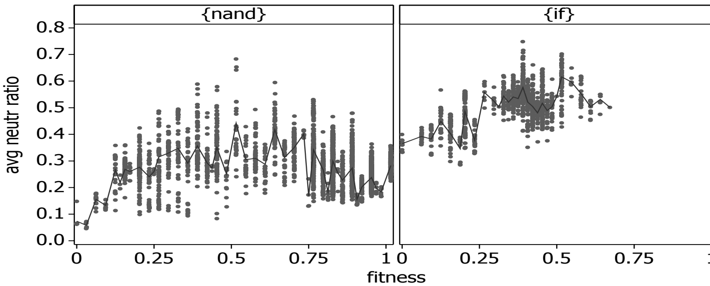


Fig. 2. Scatterplot of the average neutrality ratio in $\mathcal{L}^{\{\mathrm{NAND}\}}_{(6,6)}$ (left part) and $\mathcal{L}^{\{\mathrm{IF}\}}_{(6,5)}$ (right part)

in section 2) of all the individuals in a network. High values of $\bar{r}$ (near to 1) correspond to a large amount of neutral mutations. As figure 2 clearly shows, $\mathcal{L}^{\{\mathrm{IF}\}}_{(6,5)}$ has a higher neutrality ratio than $\mathcal{L}^{\{\mathrm{NAND}\}}_{(6,6)}$, in particular for networks at good fitness values. In other words, $\mathcal{L}^{\{\mathrm{IF}\}}_{(6,5)}$ is "more neutral" than $\mathcal{L}^{\{\mathrm{NAND}\}}_{(6,6)}$ in good regions of the fitness landscape. In this figure, as in all the subsequent ones, to guide the eye, a gray line is drawn, joining all the average points for each considered fitness value. These averages have been weighted according to the size of networks representing each point. Furthermore, points at the same coordinates have been artificially (slightly) displaced, so that they can be distinguished.

The second measure we study is the *average* Δ-*fitness* of the neutral networks. This measure is the average fitness gain (positive or negative) achieved after a mutation of

the individuals belonging to the network. Formally, let N be a neutral network, then its *average* Δ*-fitness* can be defined as:

$$\Delta\bar{f}(N) := \frac{1}{|N|} \cdot \sum_{s \in N} \left[\frac{\sum_{v \in \mathcal{V}(s)} (f(v) - f(s))}{|\mathcal{V}(s)|} \right]$$

This measure is clearly related to the notions of evolvability [1] and innovation rate [7]. It also helps to statistically describe the graph $(\mathcal{S}, \mathcal{V})$. A negative value of $\Delta\bar{f}$ corresponds to a fitness improvement (because it reduces the error) while a positive one corresponds to a worsening (because it increases the error). The average Δ-fitness scatterplots are not reported here to save space, but we have studied them and we have observed that improving good individuals for $\mathcal{L}^{\{\mathrm{IF}\}}_{(6,5)}$ is easier than for $\mathcal{L}^{\{\mathrm{NAND}\}}_{(6,6)}$, in fact, for neutral networks at good fitness values, the value of the average Δ-fitness for $\mathcal{L}^{\{\mathrm{NAND}\}}_{(6,6)}$ is positive and much larger than the one for $\mathcal{L}^{\{\mathrm{IF}\}}_{(6,5)}$.

Now, we present two measures that we have called *Non Improvable (NI) Solutions ratio* and *Non Worsenable*[1] *(NW) Solutions ratio*. The first one is defined as the number of non-improvable solutions, or non-strict local optima (i.e. individuals i which cannot generate offspring j by applying a *StSt* mutation such that the fitness of j is better than the fitness of i) that are contained into a network divided by the size of the network. The second one is the ratio of the individuals i which cannot generate offspring j (by applying a *StSt* mutation) such that the fitness of j is worse than the fitness of i. The scatterplots of *NI* solutions ratios are reported in figure 3. $\mathcal{L}^{\{\mathrm{NAND}\}}_{(6,6)}$ presents some *NI*

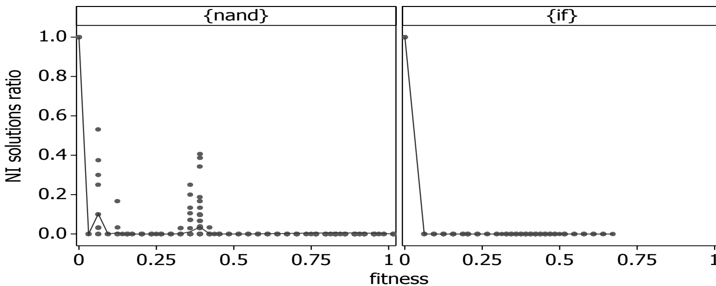


Fig. 3. Scatterplot of *NI* solutions ratio in $\mathcal{L}^{\{\mathrm{NAND}\}}_{(6,6)}$ (left part) and $\mathcal{L}^{\{\mathrm{IF}\}}_{(6,5)}$ (right part)

solutions ratios larger than 0.2 for some good fitness values (see for instance the peaks at fitness values approximately equal to 0.1, 0.125, 0.375). This indicates the presence of some trap neutral networks at this fitness values. This is not the case for $\mathcal{L}^{\{\mathrm{IF}\}}_{(6,5)}$ where *NI* solutions ratios are always equal to zero, except the obvious case of fitness equal to zero, where the *NI* solutions ratio is, of course, equal to one. In other words, for

[1] We are aware that the word "worsenable" does not exist in the English dictionary. Nevertheless we use it here as a contrary of "improvable", i.e. as something that cannot be worsened.

$\mathcal{L}^{\{NAND\}}_{(6,6)}$ some good individuals exist that cannot be improved by means of mutation, while this is not the case for $\mathcal{L}^{\{IF\}}_{(6,5)}$. *NW* solutions ratios scatterplot are not reported here to save space. Nevertheless, we have studied them and we point out that neutral networks in $\mathcal{L}^{\{IF\}}_{(6,5)}$ contain more *NW* solutions than for the ones in $\mathcal{L}^{\{NAND\}}_{(6,6)}$.

Figure 4 shows the scatterplot of unprofitable mutations ratios: for $\mathcal{L}^{\{NAND\}}_{(6,6)}$ and $\mathcal{L}^{\{IF\}}_{(6,5)}$: for each neutral network, we have calculated the number of mutations which do

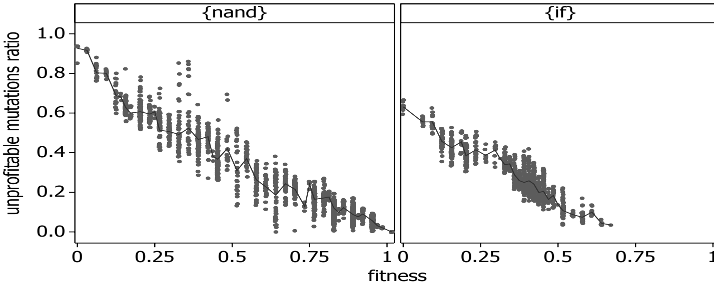


Fig. 4. Scatterplot of unprofitable mutations ratio in $\mathcal{L}^{\{NAND\}}_{(6,6)}$ (left part) and $\mathcal{L}^{\{IF\}}_{(6,5)}$ (right part)

not generate better offspring and divided it by the total number of possible mutations of the individuals in that network. Values of the unprofitable mutation ratios are higher for good fitness values in $\mathcal{L}^{\{NAND\}}_{(6,6)}$ than in $\mathcal{L}^{\{IF\}}_{(6,5)}$. In particular, for fitness values between 0 and 0.25, the majority of the possible mutations in $\mathcal{L}^{\{NAND\}}_{(6,6)}$ are unprofitable, while for $\mathcal{L}^{\{IF\}}_{(6,5)}$ only about half of the possible mutations appear to be unprofitable. All the neutrality measures that we have studied indicate that $\mathcal{L}^{\{IF\}}_{(6,5)}$ should be easier than $\mathcal{L}^{\{NAND\}}_{(6,6)}$ for GP. Furthermore, separate studies that we have done exhaustively generating all the possible individuals of the similar but smaller $\mathcal{L}^{\{IF\}}_{(3,2)}$ and $\mathcal{L}^{\{NAND\}}_{(3,3)}$ landscapes lead us to the same conclusions. Thus we hypothesize that our sampling technique is a suitable one to study our neutrality measures (i.e. the qualitative trends of our neutrality measures are kept as in the original complete landscape by our sampling technique).

4 Conclusions and Future Work

Some characteristics of fitness landscapes related to neutrality have been investigated in this paper for two different versions of the multiplexer problem. In particular, we have defined: the *average neutrality ratio*, the *average* Δ*-fitness*, the *non-improvable* and “*non-worsenable*” *solutions ratio* and the *profitable* and *unprofitable mutations ratio* of neutral networks. Each one of these measures, if considered alone, gives too a particular

vision of the fitness landscape to allow us to draw strong conclusions about its difficulty. But considered all together, they have allowed us to have a clear and rather complete picture of the characteristics of multiplexer functions landscapes. In particular, all these measures have contributed to give an interpretation of the fact that the set of operators {IF} induces an easier fitness landscape than {NAND} for the multiplexer problem. This facts have also been experimentally demonstrated by executing 100 independent GP runs for each one of these problems and calculating the success rate. As a further confirmation, we have also calculated the value of another GP hardness indicator, called Negative Slope Coefficient. Another interesting result that we have obtained with our measures is that the landscapes induced by {IF} appear to be "more neutral" than the corresponding ones induced by {NAND}, in particular in correspondance of neutral networks with good (although not optimal) fitness values. In many recent contributions, a bound between neutrality and GP performance has been hypothesized and neutrality has been presented as a profitable [14,6,22] or unprofitable [5] characteristic of fitness landscapes. What may often be misleading in these discussions is, in our opinion, *what kind* of neutrality is being considered: many different ways of intending and formalizing the concept of neutrality may exist and each one of them may lead to different, and in some cases conflicting, conclusions. Our opinion is that, to study the relationship between neutrality and difficulty of a fitness landscape, a *pool* of neutrality measures is needed. All our results considered, we argue that our measures may be helpful in studying neutrality and relate it to GP problem hardness. Results shown in this paper hold both for "small" fitness landscapes, that we have been able to study by exhaustively generating all the individuals, and for "large" fitness landscapes, obtained by increasing the problem order and the maximum size of the individuals, and that we have sampled using a new methodology. This methodology is based on a modified version of the Metropolis algorithm, enriched by two further algorithms that we have called *vertical* and *horizontal* expansion. By this strategy, it has been possible to generate and to study a large number of individuals that would not (or would very rarely) have been generated by means of a uniform random sampling or a standard Metropolis algorithm. Since our techniques are general and can be used for any GP program space, future work includes extending this kind of study to other problems and possibly defining new measures of problem hardness based on neutrality.

References

1. L. Altenberg. The evolution of evolvability in genetic programming. In K. Kinnear, editor, *Advances in Genetic Programming*, pages 47–74, Cambridge, MA, 1994. The MIT Press.
2. L. Barnett. *Evolutionary Search on Fitness Landscapes with Neutral Networks*. PhD thesis, University of Sussex, 2003.
3. P. Collard, M. Clergue, and M. Defoin-Platel. Synthetic neutrality for artificial evolution. In *Artificial Evolution*, pages 254–265, 1999.
4. P. Collard, S. Verel, and M. Clergue. Local search heuristics: Fitness cloud versus fitness landscape. In R. L. D. Mántaras and L. Saitta, editors, *2004 European Conference on Artificial Intelligence (ECAI04)*, pages 973–974, Valence, Spain, 2004. IOS Press.
5. M. Collins. Finding needles in haystacks is harder with neutrality. In H.-G. Beyer *et al.*, editor, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1613–1618, Washington DC, USA, 25-29 June 2005. ACM Press.

6. N. Geard. A comparison of neutral landscapes – nk, nkp and nkq. In *Congress on Evolutionary Computation (CEC'02)*, Honolulu, Hawaii, USA, 2002. IEEE Press, Piscataway, NJ.
7. M. Huynen. Exploring phenotype space through neutral evolution. *J. Mol. Evol., 43:165–169*, 1996.
8. J. R. Koza. *Genetic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.
9. W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer, Berlin, Heidelberg, New York, Berlin, 2002.
10. N. Madras. *Lectures on Monte Carlo Methods*. American Mathematical Society, Providence, Rhode Island, 2002.
11. C. M. Reidys and P. F. Stadler. Neutrality in fitness landscapes. *Applied Mathematics and Computation*, 117(2–3):321–350, 2001.
12. P. Schuster, W. Fontana, P. F. Stadler, and I. L. Hofacker. From sequences to shapes and back: a case study in RNA secondary structures. In *Proc. R. Soc. London B.*, volume 255, pages 279–284, 1994.
13. P. F. Stadler. Fitness landscapes. In M. Lässig and Valleriani, editors, *Biological Evolution and Statistical Physics*, volume 585 of *Lecture Notes Physics*, pages 187–207. Springer, Berlin, Heidelberg, New York, 2002.
14. M. Toussaint and C. Igel. Neutrality: A necessity for self-adaptation. In *Congress on Evolutionary Computation (CEC'02)*, pages 1354–1359, Honolulu, Hawaii, USA, 2002. IEEE Press, Piscataway, NJ.
15. L. Vanneschi. *Theory and Practice for Efficient Genetic Programming*. Ph.D. thesis, Faculty of Sciences, University of Lausanne, Switzerland, 2004.
16. L. Vanneschi, Y. Pirola, P. Collard, M. Tomassini, S. Verel, and G. Mauri. A quantitative study of neutrality in GP boolean landscapes. In M. Keijzer *et al.*, editor, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'06*, volume 1, pages 895–902. ACM Press, 2006.
17. L. Vanneschi, M. Tomassini, P. Collard, and M. Clergue. Fitness distance correlation in structural mutation genetic programming. In Ryan, C., *et al.*, editor, *Genetic Programming, 6th European Conference, EuroGP2003*, Lecture Notes in Computer Science, pages 455–464. Springer, Berlin, Heidelberg, New York, 2003.
18. L. Vanneschi, M. Tomassini, P. Collard, and S. Vérel. Negative slope coefficient. a measure to characterize genetic programming fitness landscapes. In Collet, P., *et al.*, editor, *Genetic Programming, 9th European Conference, EuroGP2006*, Lecture Notes in Computer Science, LNCS 3905, pages 178–189. Springer, Berlin, Heidelberg, New York, 2006.
19. S. Wright. The roles of mutation, inbreeding, crossbreeding and selection in evolution. In D. F. Jones, editor, *Proceedings of the Sixth International Congress on Genetics*, volume 1, pages 356–366, 1932.
20. T. Yu. "Six degrees of separation" in boolean function networks with neutrality. In R. Poli *et al.*, editor, *GECCO 2004 Workshop Proceedings*, Seattle, Washington, USA, 26-30 June 2004.
21. T. Yu and J. Miller. Neutrality and the evolvability of boolean function landscape. In J. Miller *et al.*, editor, *Proceedings of the Fourth European Conference on Genetic Programming (EuroGP-2001)*, volume 2038 of *LNCS*, pages 204–217, Lake Como, Italy, 2001. Springer, Berlin, Heidelberg, New York. Lecture notes in Computer Science vol. 2038.
22. T. Yu and J. F. Miller. Finding needles in haystacks is not hard with neutrality. In J. A. Foster *et al.*, editor, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 13–25. Springer-Verlag, 2002.

Analysing the Regularity of Genomes Using Compression and Expression Simplification

Jungseok Shin[1], Moonyoung Kang[1], R.I. (Bob) McKay[1], Xuan Nguyen[3], Tuan-Hao Hoang[2], Naoki Mori[4], and Daryl Essam[2]

[1] Structural Complexity Laboratory,
Seoul National University, Seoul 151744 Korea
[2] Australian Defence Force Academy, Campbell ACT 2601
[3] Natural Computation Group, VietNam Military Technical Academy, Hanoi
[4] Osaka Prefecture University, Osaka, Japan
shin119@yahoo.com,yerihyo@hotmail.com,hao_hth@yahoo.com,rim@cse.snu.ac.kr
mori@cs.osakafu-u.ac.jp,nxhoai@gmail.com,d.essam@adfa.edu.au
http://sc.snu.ac.kr

Abstract. We propose expression simplification and tree compression as aids in understanding the evolution of regular structure in Genetic Programming individuals. We apply the analysis to two previously-published algorithms, which aimed to promote regular and repeated structure. One relies on subtree duplication operators, the other uses repeated evaluation during a developmental process. Both successfully generated solutions to difficult problems, their success being ascribed to promotion of regular structure. Our analysis modifies this ascription: the evolution of regular structure is more complex than anticipated, and the success of the techniques may have arisen from a combination of promotion of regularity, and other, so far unidentified, effects.

Keywords: Genetic Programming, compression, simplification, ppm.

1 Introduction

Modularity has been heavily studied in Genetic Programming (GP), emphasising function-call or -expansion models based in Computer Science [1,2,3], such as Koza's Automatically Defined Functions. In biological DNA systems, modularity such as in the homeobox complex arises primarily through repetition and variation of genes [5]. This difference is reflected in terminology, where 'modularity' in Computer Science implies encapsulation and re-use, but has no such connotation in Biological Sciences. In this paper, we emphasise the biological form – repetition and variation of sub-structures (in GP terms, re-use of building blocks). To reduce confusion, we eschew the term 'modularity' from this point.

Our main idea is to use compression algorithms to estimate the regularity of genotypes. To the best of our knowledge, this has not previously been done, though there is some hint in Lathrop's work [6] relating compression depth to GP behaviour, and in Svangard and Nordin's [7] use of compression to estimate

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 251–260, 2007.
© Springer-Verlag Berlin Heidelberg 2007

similarity between, rather than self-similarity within, genotypes. Compression methods (sometimes implicitly) build a model of the structural regularities in data, then use the model to build a compressed representation of the data. Hence any regularities that are represented in the model lead to increased compression. If we measure the compression, we are implicitly measuring the extent of regularity (or at least, the extent to which that regularity matched the model).

It is well-known that GP generates redundant code [8] – expression trees which may be replaced by smaller trees with the same evaluation. In this study, we are interested in regularities both in the whole code, and in the effective part of the code. Regularities may arise in the ineffective code through a variety of mechanisms, yet have no useful effect on learning. Analogously, DNA sequences contain huge segments of repeated pairs in non-coding regions, which have no known effect on the resulting biology. An analysis focusing only on overall measures of regularity – inevitably statistically dominated by the simple repetitions – would miss the far more important regularites occurring through repetition of large segments of coding regions (genes) and equally important, their regularly structured variation. Hence we are interested in analysing the compression, not only of trees, but of their effective backbones.

We would like to simplify expression trees to the simplest equivalent form. For many function domains of interest, this is a known uncomputable problem. The best we can hope is to reduce the subtrees to a near-minimal form. For this purpose, we use the Equivalent Decision Simplification (EDS [9]) method, specialised to the arithmetic domain of our experiments.

The remainder of this section gives a brief introduction to the necessary background in compression and expression simplification. Section 2 introduces our compression-based metric for comparing the regularity of trees, and two previously-studied algorithms in which improvements in performance have been ascribed to re-use of building blocks. We report on the results of our new analyses in section 3, discussing the ramifications in section 4. In section 5, we present our conclusions about the effectiveness of compression-based methods to gain understanding of the behaviour of GP systems.

1.1 Compression

Data compression is the process of encoding information in a form that uses fewer bits than the raw data. Compression algorithms proceed in two phases, modelling the original data, and encoding the symbols using the model.

Lossless compression algorithms rely on statistical redundancy, using it to represent the sender's data more concisely, but nevertheless perfectly. Thus lossless data compression will fail if the data doesn't contain a discoverable pattern – random strings are incompressible. Conversely, if the data does contain regularity, then a compression algorithm may make use of the regularity to compress the data. This is the basis of our approach. We apply state-of-the-art tree compression algorithms, with the aim of detecting regularities – i.e. predictable, repeated structure – in the data (in this case, trees generated by GP algorithms).

String vs Tree Compression. Stochastically-based compression algorithms such as Predict by Partial Match (PPM [10] generally provide higher compression ratios than the better-known dictionary-based methods such as Ziv-Lempel [11], but at the cost of speed. Since this study was conducted off-line, speed was not a primary issue, so stochastic methods were used. XMLPPM [12] extends the PPM model to compress trees rather than strings. It does so using a stack to record the context at all branch points. XMLPPM uses XML format to represent the input trees, rather than more economical tree representations; but this is merely a technical detail. In our experiments, we record the original tree size in a direct (inorder) representation, then convert trees to XML format for compression.

1.2 Simplification

Rule-based Simplification. Previous simplification work [13,14,15] used syntactic methods to simplify trees; but finding all syntactic redundancies is undecidable. This point is not merely theoretical. If the subtreee T is complex, syntactic algorithms may find $(T-1)+(1-T)$ difficult to simplify.

Equivalent Decision Simplification. A stronger method, Equivalent Decision Simplification (EDS), has recently been proposed [9]. In EDS, a tree and its putative simplification are evaluated over a range of inputs (generally, the inputs used to define the function to be approximated). When the values of the two trees are within a predefined error bound, the two trees are judged equivalent. For this problem domain, EDS found far more simplifications than rule-based simplification. The results strongly suggested that EDS was finding essentially all the simplifications available (to be precise, the behaviour of simplified-genotype entropy, and of phenotype entropy, were virtually identical).

The problem remains, of finding an appropriate tree for substitution. We solve this by making single terminal nodes the candidates for substitution. By recursively simplifying each subtree, the whole tree is simplified. EDS is used to simplify trees in this experiment, so that we can investigate the behaviour of the effective code.

2 Methods

Using PPM-based tree compression and EDS, we re-analyse data from two previous sets of experiments which involved hypotheses about the replication of building blocks. Both were based on the TAG representation [16]. The details of the representation are unimportant here; its key properties are

- TAG representation is based on labelled trees, as in Koza-style GP
- the system uses subtree crossover and mutation, as in Koza-style GP
- Any rooted subtree of a valid TAG tree, or any consistent extension, is valid

Because of the last property (which we call 'feasibility'), it is possible to extend TAG3P in many ways. In this paper, we consider two such extensions:

- Extending TAG3P with subtree duplication and truncation operators
- Extending TAG3P with a developmental evaluation process.

2.1 Compression and Small Trees

While compression algorithms usually compress large objects well, they generally have a startup cost, with difficulty in compressing small objects. Thus relative compression of two subtrees of the same size gives useful information about their relative regularity, but can be misleading for trees of very different sizes. Instead, we report the relative compression, relative to two extremes. At one extreme, random trees are incompressible; on the other, XMLPPM is especially effective at compressing linear trees. We report the relative compressed size of an individual, relative to random and linear trees. In detail, if a tree of size S compresses to size C, we first compute the maximum compressed size R of random trees, and L of linear trees, originally of size S, and we report the ratio $(C - L)/(R - L)$. The metric has the following desirable properties:

- For trees of a given original size, it is monotonic with the compression (and by extension, estimates the degree of replication)
- It discounts any size-dependent start-up cost of the compression algorithm

The random trees were generated by half-and-half initialisation. Figure 1 shows the compressed size (Y axis) vs original size (X axis). Note the high degree of nonlinearity at small sizes, and the step in compression of random trees at an original size of between 5,000 and 10,000. The latter is probably an artefact of the fixed-size context used by PPM algorithms. An alternative algorithm, PPM*, uses unbounded context, but has never been implemented for tree-compression. We hope to report on its use in the near future.

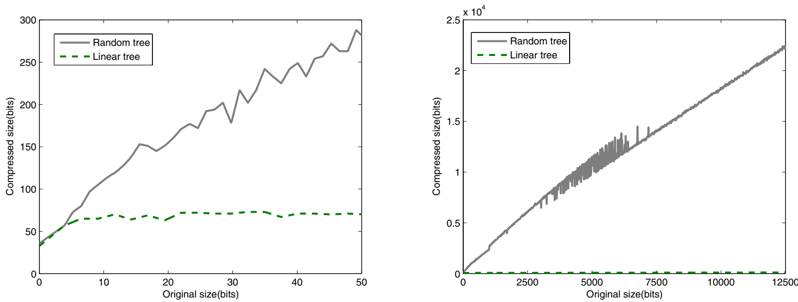


Fig. 1. Compression of Random and Linear Trees (left: small trees; right: large)

While the genotype representations considered here differ widely, all are eventually converted to a standard expression tree for evaluation. For comparability, all compression measurements are performed on expression trees, not on the primary genotypic representation.

2.2 Duplication/Truncation Operators

In the duplication operator, a random node in the current tree is chosen. The subtree rooted at that node is copied. A random leaf node in the tree is then chosen. The copied subtree is attached at that leaf node.

The duplication operator expands the size of the TAG3P tree. Used alone, it causes bloat so rapid as to prevent useful evolution. Its use is balanced by another operator, truncation. Truncation also randomly selects a location in the current tree. Instead of copying the subtree rooted at that node, truncation removes it. Duplication and truncation operators were applied at the same rate, to balance their effects on size. While their interaction with selection is difficult to analyse, in practice bloating was seen to be roughly similar to that observed in the basic TAG3P system, and typical of a tree-based GP system. Duplication was expected to be a useful operator for problems with highly regular forms, such as the well-known polynomial regression problem, in which GP is used to find a function fitting 20 points generated by a simple polynomial expression – in this case, $F_n(x) = x+x^2+x^3+...+x^n$, for various values of n. The grammar in table 1 defines the solution space. Further symbols may appear after simplification (i.e. they are not used in the evolutionary process): V is extended with a further non-terminal $CONST$, and T with two further terminals, $0, 1$, while P is extended with two additional productions: $EXP \rightarrow CONST$ and $CONST \rightarrow 0|1$.

Table 1. Grammar describing Solution Space

$G = V, T, P, S$	$EXP \rightarrow EXP\ OP\ EXP \| sin\ EXP \| VAR$
$S = EXP$	$OP \rightarrow + \| - \| * \| /$
$V = EXP, OP, VAR$	$VAR \rightarrow x$
$T = x, sin, +, -, *, /$	

Our expectations were borne out in experiments reported in [16]. Duplication improved performance when used as a mutation operator in TAG3P, and more substantially when used as a local search operator. However the underlying assumption, that use of duplication had promoted duplicated code segments appropriate for the domain, was not tested. We compared five algorithms (TAG: the basic TAG3P system; TAGCROSS: TAG3P with no mutation operator; TAGM: TAG3P with duplication/truncation as mutation operators; LSTAG10 and LSTAG50: TAG3P with duplication/truncation as local search operators – 10 and 50 steps respectively). All algorithms had a fixed budget of evaluations, so more local search corresponded to fewer evolutionary generations.

2.3 Developmental Evaluation

Developmental evaluation [17] has been proposed as a mechanism to promote structural regularity in GP. It uses a typical developmental process, but relying on the feasibility property of TAG3P, evaluates and selects individuals during development, in analogy with biological systems. In more detail, individuals are evaluated on a family of problems of increasing difficulty as they develop, with performance at earlier stages favoured over performance at later stages. In experiments using the family of functions $F_1, ...F_9$, and the grammar from table 1, developmental systems achieved excellent performance (two variants

were used, DEVTAG using a trivial incremental growth process, and DTAG3P, an L-systems developmental model; in the experiments, they were compared with standard tree-based GP (DE), TAG3P, and DTAG3PF9ALL: using DTAG3P L-system representation, but standard evolutionary evaluation). Recently [18], we reported on comparisons between these systems using direct compression measurements, and noted the difficulty of comparing compression ratios for trees of differing sizes. We overcome this with the ratio method from section 2.1.

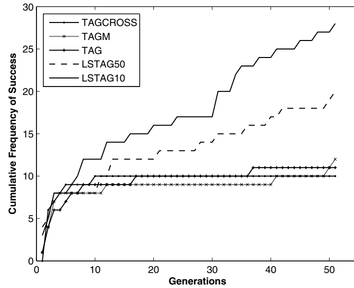


Fig. 2. Cumulative Success Ratio vs Evaluations ($*10^4$)

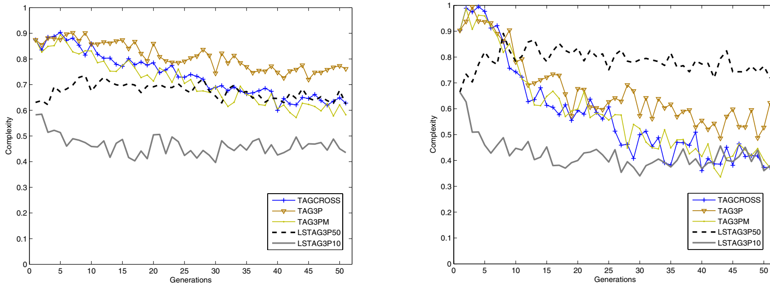


Fig. 3. Duplication: Compression Complexity vs Evaluations ($*10^4$) (left: original; right: effective code)

3 Results

3.1 Duplication/Truncation

Figure 2 shows the success ratio (out of 100 runs) for the different algorithms. As previously reported, using crossover only, or duplication/truncation as mutation operators, makes little difference to the overall performance of the underlying TAG3P algorithm (which performs slightly better than standard GP). Local search for 10 steps of duplication and truncation substantially improves performance, but too much local search degrades it.

Figure 3 shows the complexity of the fittest individual in each generation, using our measure. We note that individual complexity tends to decrease throughout a run, both for the original code and the effective backbone; and that 10 steps of local search give the lowest complexity for either original or effective code, but 50 steps give the highest for the latter.

3.2 Developmental Evaluation

Previously presented results have shown the very high success rates of the DTAG3P and DEVTAG algorithms relative to the underlying TAG algorithm on this problem (around 70% and 30% success respectively, for the same number of potential evaluations as used in subsection 3.1). For the same budget of evaluations, standard tree-based GP has essentially zero probability of success.

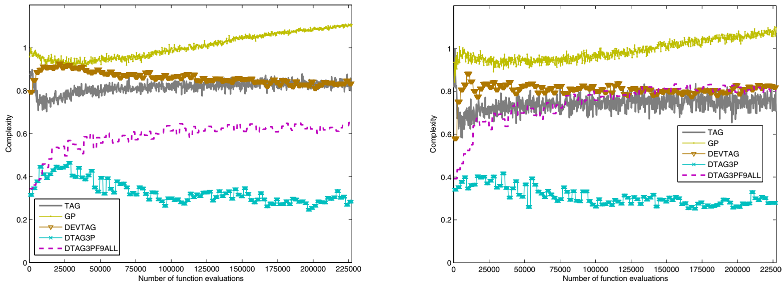


Fig. 4. Developmental Evaluation: Compression Complexity vs Generations (left: original; right: effective code)

Figure 4 shows the compression-complexity of the algorithms plotted against generations. We note the very low replication in both overall and effective code components of standard GP; and the high high replication in both overall and effective code components of DTAG3P. As far as code complexity is concerned, the DEVTAG algorithm performs almost identically to the underlying TAG3P algorithm. Finally, L-system development on its own (DTAG3PF9ALL) promotes code replication at the overall code level, but without developmental evaluation, this replication is not reflected in the effective code.

4 Discussion

4.1 Duplication/Truncation

The results are partially consistent with our original hypothesis, that code replication would help to solve the F_9 problem. 10 steps of local search with duplication and truncation improve the probability of success, and increase the amount of code repetition, both in original and effective code. But this is not the whole story. 50 steps of local search degrade the effectiveness of the algorithm, but

not so far as to be worse than the non-local-search versions (perhaps 50 steps of local search do not allow enough generations of evolutionary search for effective exploration). Yet 50 steps of local search give the lowest level of effective code replication. Thus success is not simply correlated with replicated effective code. This leads on to a range of potential hypotheses. Perhaps truncation is more important to the success of the success of the duplication/truncation local search combination than we had supposed. Perhaps what is replicated is more important than the degree of replication. Compression analysis has opened up a range of new questions about this matter.

4.2 Developmental Evaluation

Similarly, the Developmental Evaluation results strongly support the underlying hypothesis, that a developmental process, and evaluation during development, are both necessary for promoting repetitively structured effective code, such as arises in DNA. Interestingly, this contrasts strongly with our earlier results (using simple compression ratios, rather than the more size-independent method used here), which seemed to disprove this hypothesis. L-system-based development alone, without developmental evaluation (DTAG3PF9ALL), initially generates a high degree of regularity, but this regularity is rapidly lost from the effective code, and more gradually from the overall code.

However this is not the whole story. DEVTAG (which uses a trivial developmental process, but still uses incremental evaluation during development) performs well on this problem – far better than DTAG3PF9ALL – yet incremental evaluation does not promote regularity at all in this case (the individuals are, if anything, less regular than those generated by the basic TAG algorithm).

5 Conclusions

In sections 4.1 and 4.2, previously accepted hypotheses about the mode of action of particular evolutionary systems were tested using the twin tools of EDS and compression. In both cases, in their broadest outlines, the hypotheses were borne out by these analyses. In more detail, they raised intriguing new questions. In both cases, the success of the proposed modifications does seem to reflect increased regularity of the effective code; however this is not the whole story, and the effectiveness of the systems is due, in part, to causes other than simple replication of good building blocks. This has opened up new lines of inquiry, investigating what has allowed these systems to perform so effectively.

We don't believe this situation is unique to these two specific problems. These techniques can help to understanding the behaviour of evolutionary algorithms in any domain where replication of building blocks may be important (and in particular, to understanding the level of success achieved by modularising approaches such as ADFs). But we don't believe the applications stop there. In these experiments, we compressed individual trees, to determine the level of replication within trees. It is equally straightforward to compress whole populations, and use this compression to estimate the level of replication of code

segments across the population. Such analyses should allow us to gain a deeper understanding of the role of building blocks, since any promotion of building blocks should result in regularities in effective code that compression algorithms should be able to discover and exploit.

The compression techniques are not problem- or representation- specific, so they are readily extended to new systems and problem domains. The principle of EDS is equally extensible, at least to problem domains where a reasonable definition of 'equivalent decision' is available – for example to other arithmetic function sets. However in some domains, it may be unnecessary. in Boolean domains, EDS reduces to model-theoretic (i.e. truth- table) methods, and syntactic simplification to proof-theoretic methods. For perfect accuracy, the former are exponential time, while the latter are NP-complete. It is not yet clear which will yield better performance if some level of missed equivalences is acceptable.

Acknowledgements. This research was financially supported by a Seoul National University support grant for new faculty.

References

1. Koza, J.R.: Hierarchical automatic function definition in genetic programming. In Whitley, L.D., ed.: Foundations of Genetic Algorithms 2, Vail, Colorado, USA, Morgan Kaufmann (24–29 July 1992) 297–318
2. Spector, L.: Evolving control structures with automatically defined macros. In Siegel, E.V., Koza, J.R., eds.: Working Notes for the AAAI Symposium on Genetic Programming, MIT, Cambridge, MA, USA, AAAI (10–12 November 1995) 99–105
3. Woodward, J.R.: Modularity in genetic programming. In Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., Costa, E., eds.: Genetic Programming, Proceedings of EuroGP'2003. Volume 2610 of LNCS., Essex, Springer-Verlag (14-16 April 2003) 254–263
4. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge Massachusetts (May 1994)
5. Schlosser, G., Wagner, G.e.: Modularity in Development and Evolution. University of Chicago Press, Chicago, Ill, USA (2004)
6. Lathrop, J.I.: Compression depth and genetic programs. In Koza, J.R., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M., Iba, H., Riolo, R.L., eds.: Genetic Programming 1997: Proceedings of the Second Annual Conference, Stanford University, CA, USA, Morgan Kaufmann (13-16 July 1997) 370–379
7. Svangard, N., Nordin, P.: Automated aesthetic selection of evolutionary art by distance based classification of genomes and phenomes using the universal similarity metric. In Raidl, G.R., Cagnoni, S., Branke, J., Corne, D.W., Drechsler, R., Jin, Y., Johnson, C.R., Machado, P., Marchiori, E., Rothlauf, F., Smith, G.D., Squillero, G., eds.: Applications of Evolutionary Computing, EvoWorkshops2004: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoMUSART, EvoSTOC. Volume 3005 of LNCS., Coimbra, Portugal, Springer Verlag (5-7 April 2004) 447–456
8. Blickle, T., Thiele, L.: Genetic programming and redundancy. In Hopf, J., ed.: Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken), Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany, Max-Planck-Institut für Informatik (MPI-I-94-241) (1994) 33–38

9. Mori, N., McKay, R., Nguyen, X., Essam, D.: How different are genetic programs: New methods for studying diversity and complexity in genetic programming. in preparation (2007)
10. Cleary, J., Witten, I.: Data compression using adaptive coding and partial string matching. IEEE Transactions on Communications **32** (1984) 396–402
11. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory **24** (1978) 530–536
12. Cheney, J.: Compressing xml with multiplexed hierarchical models. In: IEEE Data Compression Conference, Snowbird, Utah (2002) 163–172
13. Hooper, D., Flann, N.S.: Improving the accuracy and robustness of genetic programming through expression simplification. In Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L., eds.: Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, MIT Press (28–31 July 1996) 428
14. Ekart, A.: Shorter fitness preserving genetic programs. In Fonlupt, C., Hao, J.K., Lutton, E., Ronald, E., Schoenauer, M., eds.: Artificial Evolution. 4th European Conference, AE'99, Selected Papers. Volume 1829 of LNCS., Dunkerque, France (3-5 November 2000) 73–83
15. Wong, P., Zhang, M.: Algebraic simplification of genetic programs during evolution. Technical Report CS-TR-06-7, Computer Science, Victoria University of Wellington, New Zealand (2006)
16. Hoai, N.X., McKay, R.I., Essam, D., Hao, H.T.: Genetic transposition in tree-adjoining grammar guided genetic programming: The duplication operator. In Keijzer, M., Tettamanzi, A., Collet, P., van Hemert, J.I., Tomassini, M., eds.: Proceedings of the 8th European Conference on Genetic Programming. Volume 3447 of Lecture Notes in Computer Science., Lausanne, Switzerland, Springer (30 March - 1 April 2005) 108–119
17. Hoang, T., Essam, D., McKay, R., Nguyen, X.: Developmental evaluation in genetic programming: A tag-based framework. In: Procceedings of the third Asia-Pacific Workshop on Genetic Programming, VietNam Military Technical Academy, Hanoi, VietNam (October 12-14 2006)
18. Kang, M., Shin, J., Hoang, T., McKay, R., Essam, D., Mori, N., Nguyen, X.: Code duplication and developmental evaluation in genetic programming. In: 10th Asia-Pacific Workshop on Intelligent and Evolutionary Systems, Seoul, Korea (2006)

Changing the Genospace: Solving GA Problems with Cartesian Genetic Programming

James Alfred Walker and Julian Francis Miller

Intelligent Systems Group, Department of Electronics, University of York, Heslington, York, YO10 5DD, UK
{jaw500,jfm7}@ohm.york.ac.uk

Abstract. Embedded Cartesian Genetic Programming (ECGP) is an extension of Cartesian Genetic Programming (CGP) capable of acquiring, evolving and re-using partial solutions. In this paper, we apply for the first time CGP and ECGP to the ones-max and order-3 deceptive problems, which are normally associated with Genetic Algorithms. Our approach uses CGP and ECGP to evolve a sequence of commands for a tape-head, which produces an arbitrary length binary string on a piece of tape. Computational effort figures are calculated for CGP and ECGP and our results compare favourably with those of Genetic Algorithms.

1 Introduction

Embedded Cartesian Genetic Programming (ECGP) is an extension of Cartesian Genetic Programming (CGP) incorporating ideas from Module Acquisition [1], which allows the automatic acquisition, evolution and re-use of partial solutions in the form of modules. Previous work [2] has shown ECGP to be more computationally efficient than CGP on a range of digital circuit problems and the speedup grows with problem difficulty.

Recently, CGP and ECGP have been applied to the Genetic Algorithm (GA) based Hierarchical-if-and-only-if (H-IFF) problem [3]. CGP and ECGP found solutions to the H-IFF problem more easily than published attempts using a GA. This paper builds on the work from [3] by applying the same technique to two GA benchmarks; the one-max problem and the order-3 deceptive problem.

The one-max problem [4] was originally used to test the generality of hill-climbing search algorithms but is now more commonly used as a GA benchmark [5]. The objective of the problem is to find a binary string of length n, which contains all 1's. The order-3 deceptive problem was proposed by Goldberg [6] and has also been widely adopted as a challenging problem for GAs. The problem analyses similarities in a binary string using 3-bit schemata. The aim of the problem is to find a binary string containing all 1's, therefore consisting only of the 3-bit schema containing all 1's. This schema is associated with the highest fitness. The only other fitness rewards are associated with schemata containing all 0's or a single 1. This leads the GA away from the global optimum and towards the global minimum, and is the reason why the problem is classed as deceptive.

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 261–270, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Some work already exists on evolving GAs using alternate forms of evolutionary computation. Miller and Yu [7] implemented a form of CGP to evolve binary strings to the one-max problem when they were investigating the properties and utility of neutrality. Unlike this approach, in this paper the link between the number of nodes encoded in the genotype and length of the binary string is indirect and uncorrelated, we anticipate that this will allowing better scaling on large problem instances. Ryan et al [8] have developed a technique called GAuGE, which extends Grammatical Evolution (GE) to form a position independent GA. GAuGE has been applied to the one-max problem and an extension of GAuGE called LINKGAuGE, which employs tight linkage between the genes of the GA [9], has been applied to the order-3 deceptive problem. Another GE based approach is the meta-Grammar Genetic Algorithm (mGGA) [10], which allows the construction of small bit-strings that are re-used when forming the solution bit-string.

The plan for the paper is as follows: Sections 2 and 3 give an overview of CGP and ECGP. In section 4, we describe our approach of applying CGP and ECGP to GA problems. The details of our experiments are shown in section 5 followed by the results and comparisons in section 6. Section 7 gives conclusions and suggestions for future work.

2 Cartesian Genetic Programming (CGP)

CGP is a form of GP based on acyclic directed graphs, which is only modified by mutation [11]. CGP uses a fixed length representation, where the genotype consists of a list of integers, encoding the function and connections of each node in the directed graph. However, the number of nodes in the directed graph (phenotype) can vary but is bounded, as every node encoded in the genotype does not have to be connected. This allows areas of the genotype to be inactive and have no influence on the phenotype, leading to a neutral effect on genotype fitness called neutrality. This unique type of neutrality has been investigated in detail [11] and found to be extremely beneficial to the evolutionary process on the problems studied.

Each node in the directed graph is encoded in the genotype by a number of genes, determined by the arity of the function the node represents. For each encoded node, the first gene encodes the node's function (using values from a lookup table) and the remaining genes encode the node's input connections (using the index label of the node or program input). The nodes take their inputs in a feed forward manner from either the output of a previous node in the directed graph or from the program inputs (terminals). The program inputs are labelled from 0 to n-1 where n is the number of program inputs. The nodes in the directed graph are also labelled sequentially starting from n to n+m-1 where m is the number of nodes in the directed graph. If the problem requires k program outputs then k integers are added to the end of the genotype, each one encoding the index of the node in the directed graph where the program output is taken from. These k integers are initially set as pointers to the outputs of the

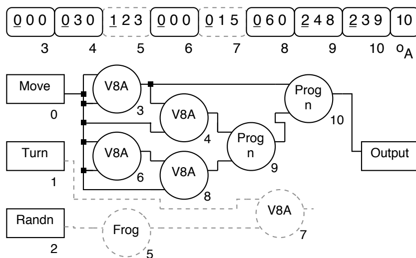


Fig. 1. CGP genotype and corresponding phenotype for the 8-bit one-max problem. The underlined genes encode the function of each node using the lookup table: V8A(0), Frog(1), Progn(2). See Section 4 for function details. The index labels are shown underneath each program input and node. The inactive areas of the genotype and phenotype are shown in grey dashes.

last k nodes encoded in the genotype. Fig. 1 shows a CGP genotype and how it is decoded for the 8-bit one-max problem.

3 Embedded Cartesian Genetic Programming (ECGP)

ECGP incorporates ideas from Module Acquisition [1] with CGP, to allow the automatic acquisition, evolution and re-use of partial solutions (referred to as modules) [2]. Thereby giving CGP a form of Automatically Defined Function (ADF) [12]. This paper only gives a brief overview of ECGP due to space restrictions. For information on the technical details of ECGP, please refer to [2].

ECGP uses a modified CGP *genotype* making it a bounded variable length representation (in terms of the number of encoded nodes in the genotype and the number of genes used to encode each node). The number of nodes encoded in the genotype decreases when sections of the genotype are encapsulated into modules (when modules are created by the compress operator) and increases when modules are expanded back into sections of the genotype (when modules are destroyed by the expand operator). The number of genes used to encode the inputs of a node in the genotype can vary as a result of either module mutation increasing or decreasing the number of module inputs (therefore affecting the number of genes required to encode the module), or a module being introduced into the genotype (requiring extra genes to encode all of the module inputs).

Modules are capable of having multiple outputs, but the CGP representation only encodes nodes with single outputs, therefore each gene is now represented using a pair of integers rather than just a single integer, as in CGP. For each gene encoding a node input, the first integer encodes the node index (as in CGP), whilst the second integer encodes the function output used.

Using a pair of integers to encode each function gene allows the introduction of node types into the ECGP representation. Node types allow the identification of nodes encoded in the genotype representing: primitive functions (node type 0), modules that contain an original section of the genotype (node type I) and

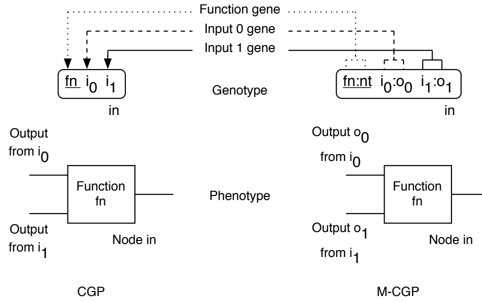


Fig. 2. CGP and ECGP genotypes and corresponding phenotypes for a single node. The components of each gene are labelled as follows: function (*fn*), node type (*nt*), node indexes that the node inputs are taken from (i_0, i_1), node outputs that the node inputs are taken from (o_0, o_1), index of this node (*in*).

modules that contain a re-used section of the genotype (node type II). Operators act differently on the nodes encoded in the genotype depending on their node type. Node types are encoded as the second integer of the function gene of every node, the first integer encodes the primitive function (as in CGP) or module (using values from a lookup table). Figure 2 illustrates the differences between the CGP and ECGP representations.

Modules are represented using a modified ECGP representation, which also encodes structural information about the module. Four extra integers are added to the beginning of the module genotype to encode the module identifier, the number of inputs and outputs of the module, and the number of nodes the module contains. Currently, a module can only contain nodes of type 0, to prevent bloat inside the module. Once a module is created, it is added to the module list (a dynamic extension of the function list) and can be re-used whilst the module remains in the module list, along with the primitive functions. The module list is updated every generation to contain the module list of the fittest individual in the population (in accordance with the 1 + 4 evolutionary strategy).

The module genotype can be evolved by the module mutation operators independently of the ECGP genotype. Either a structural mutation can occur, which affects the number of module inputs and outputs, or a point-mutation can occur, which affects the nodes contained in the module (as mutation would occur in CGP).

4 Applying CGP and ECGP to GAs

One of the main issues faced was deciding how to apply CGP to GA problems. A method was needed which would scale well for different length bit strings and would not require changes to the number or type of program inputs. The method chosen in this paper was heavily influenced by a GP benchmark problem called the Lawnmower Problem [12]. In the lawnmower problem, GP is used to evolve

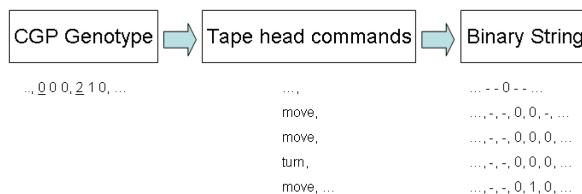


Fig. 3. The three step procedure for producing a GA bit-string from the CGP genotype, via a set of tape head commands

a set of commands to move a lawnmower around a lawn, which has been divided into a n x m grid of squares, where n and m denote the width and height of the lawn respectively. The lawnmower cuts the grass in each square it visits, with a solution being found when the lawnmower has visited every square of the grid, therefore cutting all the grass.

In this paper, instead of evolving a set of instructions for a lawnmower on a 2-dimensional lawn, a set of commands for a moving a tape head on a 1-dimensional tape is evolved. Similar to the lawn in the lawnmower problem, the tape is divided into n squares, where n is the number of bits in the GA. Initially, all squares on the tape are blank, and the tape head is positioned in the centre square of the tape (similar to the lawnmower starting in the centre square of the lawn). In a single command, the tape head can move one square or jump a number of squares in the direction the tape head is facing (left or right). If the tape head moves off one end of the tape, it re-appears in the square at the opposite end of the tape (just as the lawnmower would in the lawnmower problem). When the tape head visits a square, the value of the square is changed according to the rule:

$$if(square == blank \mid\mid square == 1),\, square = 0 \qquad (1)$$
$$if(square == 0),\, square = 1$$

Therefore, the tape head behaves like the bit-flip operator found in GAs, once a tape square does not contain a blank. Once the set of commands has been executed, the tape head will have produced a bit-string of length n containing the symbols: - (blank), 0 and 1, which can be evaluated as an individual in a GA. A blank (-) in the bit-string does not contribute towards the fitness score, as we only want to generate bit-strings containing 0's and 1's. An example of the approach is shown in Figure 3.

Although the proposed approach changes the nature of the GA test problems, it does allow us to investigate whether the proposed approach can evolve solutions to GAs whilst taking advantage of the benefits of CGP, such as neutral drift. We believe changing the dimensionality and neutral interconnectedness of the genotype space may alleviate problems typical of GAs - early convergence on sub-optima. Due to the nature of the approach, small changes to the genotype

Table 1. The parameter settings used for CGP and ECGP (* - ECGP only). The mutation rate is expressed as a percentage of the genotype length. The operator rates and probabilities are per generation and taken from [2].

Parameter	Value
Population size	5
Genotype point mutation rate	3% (18 Genes)
Compress/Expand probability *	0.1/0.2
Module point mutation probability *	0.04
Add/Remove input probability *	0.01/0.02
Add/Remove output probability *	0.01/0.02
Maximum module size *	3 or 5 nodes
Number of independent runs	50

can produce a big change in the bit-string produced on the tape. Thereby acting like an implicitly defined variable rate mutation operator, which could reduce the mutation parameter sensitivity associated with GAs.

The CGP program has three program inputs, which are constrained versions of those used in the lawnmower problem: *move* - moves the tape head one square in the direction it is facing and changes the value of the new square according to Equation 1, *turn* - alters the direction the tape head travels along the tape from right to left or vice versa and *random constant* - a random number, r, chosen at the start of each independent run, where $0 <= r < n$. Both move and turn also return a constant, 0, so mathematical operations can also be performed on the program inputs.

The function set used contains the same functions as the lawnmower problem: *progn* - a program node, which executes the graph connected to its first input, followed by the graph connected to its second input and returns the result of the second input, *v8a* - performs addition on its two inputs and returns the result, and *frog* - moves the tape head by a number of squares specified by its input in the direction it is facing and changes the value of the new square according to Equation 1.

5 Experiment Details

The parameter settings used for CGP and ECGP on the 100, 250, 500, 1000, 2000 and 4000-bit one-max problem (using 100 nodes) and the 30-bit order-3 deceptive problem (using 25, 50, 75 and 100 nodes) are shown in Table 1.

The fitness functions used for both problems are the same as those used by GA researchers. For the one-max problem, the fitness function is the total number of ones present in the bit-string and for the order-3 deceptive problem, the fitness function is defined by the schema from [6] shown in Table 2. Any schema containing a blank(-) is awarded a fitness score of zero.

Table 2. The schema for the order-3 deceptive problem and their fitness values

String	000	001	010	011	100	101	110	111
Fitness	28	26	22	0	14	0	0	30

6 Results

Computational effort was calculated using the formula from [3] and shown in Equation 2 with $z = 99\%$. The computational effort figures for CGP and ECGP applied to the one-max and order-3 deceptive problems are shown in Tables 3 and 4 respectively. Various run statistics are also included in both tables to allow comparisons with other techniques and to illustrate how computational effort is a better measure to use than the average number of fitness evaluations, as it is more resilient to outliers in the data. A modified standard deviation is used, as the results are not normally distributed. The modified standard deviation is a statistic in which 68% of all the results lie either side of the mean.

$$P(M,i) = \frac{N_s(i)}{N_{total}},\ R(z) = ceil\left(\frac{\log(1-z)}{\log(1-P(M,i))}\right),\ I(M,i,z) = MR(z)\,i+1 \tag{2}$$

Table 3. Computational effort (CE) figures and various statistics for CGP and ECGP applied to the one-max problem for bit-strings of various lengths (NB). The statistics gathered include: average number of evaluations (AE), modified standard deviation (MSD), the quartiles (Q0-Q4), the limits for mild and extreme outliers (MO and EO) and the number of each outlier present in the data is shown in brackets.

	NB	AE	MSD	Q0	Q1	Q2	Q3	Q4	MO	EO	CE
CGP	100	1,684	1,143	197	583	895	1,906	10,405	3,891(3)	5,875(2)	5,766
	250	2,175	1,598	329	659	981	1,876	13,237	3,702(8)	5,527(5)	6,405
	500	4,850	4,074	321	869	1,471	5,026	47,013	11,262(4)	17,497(3)	9,606
	1000	2,006	1,293	441	818	1,071	1,592	23,405	2,753(6)	3,914(4)	6,120
	2000	2,146	1,165	493	1,145	1,455	2,232	15,989	3,863(5)	5,493(3)	7,203
	4000	3,340	2,312	593	1,193	1,377	2,186	33,417	3,676(8)	5,165(7)	7,203
ECGP(3)	100	5,695	5,210	201	703	1,561	6,645	45,125	15,558(4)	24,471(2)	9,610
	250	8,411	7,750	237	1,169	2,589	11,758	71,745	27,642(4)	43,525(2)	16,326
	500	6,505	5,117	541	1,405	2,443	5,209	68,429	10,915(5)	16,621(4)	16,326
	1000	39,529	37,797	637	1,700	4,665	12,873	1,290,565	29,633(5)	46,392(4)	24,010
	2000	14,186	12,151	793	2,101	3,955	10,539	169,021	23,196(7)	35,853(4)	26,888
	4000	15,125	12,592	445	2,594	5,227	11,012	256,661	23,639(5)	36,266(4)	30,728
ECGP(5)	100	11,472	10,807	301	675	1,593	5,990	220,325	13,963(6)	21,935(4)	10,248
	250	16,839	15,762	353	1,254	2,487	10,378	282,553	24,064(8)	37,750(5)	15,368
	500	14,061	12,787	701	1,425	3,333	7,778	183,753	17,308(7)	26,837(5)	20,810
	1000	22,024	19,676	613	2,064	3,461	7,794	798,149	16,389(5)	24,984(3)	23,527
	2000	19,139	17,327	661	2,329	5,407	14,366	129,725	32,422(8)	50,477(6)	32,652
	4000	19,417	15,612	873	3,866	6,469	15,577	248,457	33,144(5)	50,710(4)	42,248

Table 4. Computational effort (CE) figures and various statistics for CGP and ECGP applied to the 30-bit order-3 deceptive problem for various genotype lengths (ND). The statistics gathered include: average number of evaluations (AE), modified standard deviation (MSD), the quartiles (Q0-Q4), the limits for mild and extreme outliers (MO and EO) and the number of each outlier present in the data is shown in brackets.

	ND	AE	MSD	Q0	Q1	Q2	Q3	Q4	MO	EO	CE
CGP	25	3,814	2,898	221	1,078	2,537	5,445	18,697	12,021(4)	18,586(1)	14,088
	50	5,998	4,874	249	1,139	2,357	5,032	72,717	10,872(4)	16,711(3)	15,368
	75	118,649	117,272	153	1,329	2,535	7,600	3,918,661	17,007(9)	26,413(7)	16,648
	100	279,444	278,372	173	1,171	3,037	10,882	9,066,769	25,449(8)	40,015(8)	15,219
ECGP(3)	25	10,313	9,176	261	1,036	1,859	5,172	303,133	11,376(6)	17,580(4)	12,005
	50	48,374	47,126	401	1,266	2,715	8,049	1,390,337	18,224(9)	28,398(7)	16,648
	75	14,571	13,782	201	912	2,155	5,569	301,137	12,555(6)	19,540(6)	12,808
	100	64,164	62,735	129	1,629	2,691	7,645	1,315,681	16,669(7)	25,693(6)	15,364
ECGP(5)	25	21,548	20,132	385	1,421	2,567	4,755	681,717	9,756(8)	14,757(7)	14,724
	50	70,738	69,797	329	899	2,845	8,486	1,572,621	19,867(8)	31,247(6)	14,415
	75	16,314	15,649	233	696	1,629	13,016	255,513	31,496(7)	49,976(4)	10,413
	100	44,965	43,612	205	1,495	3,651	10,893	1,169,005	24,990(7)	39,087(7)	19,208

For both problems, all fifty independent runs of CGP and ECGP produced 100% successful solutions.

The computational effort figures for the one-max problem show CGP performs better than ECGP regardless of the maximum module size, for all lengths of bit-string. As the length of the bit-string increases, the computational effort required by CGP increases only slightly, indicating that CGP scales particularly well with problem difficulty. This suggests that CGP may perform comparatively better on larger bit-strings. In ECGP, the automatic acquisition, evolution and re-use of modules could be hindering the search performance, possibly due to a lack of modularity in the problem. Alternatively, the problem could be too simple, so by the time a useful module has been discovered, CGP has already found a solution to the problem.

The results in Table 4 show the computational effort figures for CGP and ECGP are similar, as the number of nodes increases but ECGP is capable of performing better than CGP, depending on the maximum module size chosen. This suggests ECGP is exploiting any modularity in the problem that makes it less susceptible to deception, such as the re-use of a module that creates the schema containing three ones. However, the average number of evaluations figures contradict the computational effort figures on a number of occasions. On analysis, the quartiles, Q0-Q3, for CGP and ECGP show a similar trend to the computational effort figures. However, the quartile, Q4, is quite erratic as it contains numerous mild and extreme outliers. The outliers are the reason for the contradiction between the average number of evaluation and computational effort figures, therefore showing computational effort is less influenced by the presence of outliers.

In general, the computational effort figures for CGP and ECGP increase with the number of nodes, suggesting using smaller genotypes produces better results.

Table 5. The average number of evaluations (AE) for other techniques applied to the 100-bit One-Max and 30-bit Order-3 Deceptive Problems

	100-bit One-Max			30-bit Order-3 Deceptive		
Technique	Gen-GA	Simple-GA	GAuGE	Gen-GA	Messy-GA	LinkGAuGE
AE	7,714	4,000	4,000	4,484	10,000	20,000

The larger the genotype, the longer the list of commands for the tape head. Therefore, a small change in a large genotype could drastically alter the number of commands for the tape head, making it harder to find a solution when you are only a few bits away.

The results of CGP and ECGP for the two problems are compared with other techniques found in Table 5. The generational-GA results were taken from [5], the simple-GA and GAuGE results are approximated from [8] and the messy-GA and LinkGAuGE results are approximated from [9].

For the 100-bit One-Max problem, CGP performs better than the other three techniques and ECGP (with a maximum module size of 3) performs better than the generational-GA but worse than the simple-GA and GAuGE. However, it is notable that CGP also solves the 4000-bit One-Max problem slightly faster than the simple-GA and GAuGE on the 100-bit One-Max problem.

Comparing the results of CGP and ECGP (with 25 nodes) and the other techniques on the 30-bit Order-3 Deceptive problem, once again CGP performs better than the other three techniques and ECGP performs better than LinkGAuGE, and has approximately equal performance to the messy-GA but is worse than the generational-GA.

Out of curiosity, the CGP and ECGP solutions found to the One-max problem were applied to the one-max problem with different length of bit-strings than those used to evolve the solution. The results showed that the majority of the solutions found on the original problem solved the One-max problem for all lengths of bit-string from 1-bit up to the length it was originally trained on, and also on some longer bit-strings. In one case, a CGP solution to the 100-bit One-max problem solved all One-max problems up to a length of 264-bits. This implies the solution had learned something about the form of the general solution to the One-Max problem. This was also noticed with the solutions to the order-3 deceptive problem, except that the original solution either solved all the order-3 deceptive problems up to a length of 30-bits, or it solved the order-3 deceptive problems that were a factor of the 30-bit problem, such as the 3, 6 and 15-bit problems. We intend to investigate this further in future work.

7 Conclusion

We have presented the application of CGP and ECGP to two classic GA problems: the one-max and order-3 deceptive problems. CGP was shown to perform better than ECGP on the one-max problem for various length bit-strings and was

also shown to scale well with problem difficulty. The performance of CGP and ECGP was similar on the order-3 deceptive problem, however ECGP is capable of performing better than CGP but is dependant on the relationship between the maximum module size chosen and genotype length. Comparing CGP, a simple GA, a generational GA and GAuGE on the one-max problem showed CGP to perform the best and to scale better on problem size than the others. Comparing CGP, a generational GA, a messy-GA and LINKGAuGE on the order-3 deceptive problem also showed CGP to perform the best. This could possibly indicate that the method employed in this paper not only drastically alters the search space but also takes advantage of the benefits associated with CGP (such as neutral drift) and transfers them to the GA.

Preliminary results for initialising the tape with different values (all 0's or 0's and 1's at random) have shown a decrease in the performance of CGP and ECGP, and will be investigated further in future work. It would be interesting to see if the approach described in this paper can be modified to produce floating point numbers and be applied to real-valued optimisation problems associated with classical evolutionary programming. This approach could also be used in a variety of real-world problems, such as protein folding and protein sequence comparison from the field of bioinformatics.

References

1. Angeline, P.J., Pollack, J.: Evolutionary module acquisition. In: Proc. of the 2nd Annual Conference on Evolutionary Programming. (1993) 154–163
2. Walker, J.A., Miller, J.F.: Investigating the performance of module acquisition in cartesian genetic programming. In: Proc. of GECCO. Volume 2., ACM (2005) 1649–1656
3. Walker, J.A., Miller, J.F.: Embedded cartesian genetic programming and the lawnmower and hierarchical-if-and-only-if problems. In: Proc. of GECCO, ACM (2006)
4. Ackley, D.H.: A connectionist Machine for Genetic Hillclimbing. Kluwer (1987)
5. Tuson, A., Ross, P.: Adapting operator settings in genetic algorithms. Evolutionary Computation **6**(2) (1998)
6. Goldberg, D.E., Deb, K., Korb, B.: Messy genetic algortihms: Motivation, analysis and first results. Complex Systems **3**(5) (1989)
7. Yu, T., Miller, J.F.: The role of neutral and adaptive mutation in an evolutionary search on the onemax problem. In: Late Breaking Papers at GECCO, AAAI (2002) 512–519
8. Ryan, C., Nicolau, M., O'Neill, M.: Genetic algorithms uing grammatical evolution. In: Proc. of the 5th EuroGP. Volume 2278 of LNCS., Springer (2002) 278–287
9. Nicolau, M., Ryan, C.: Linkgauge: Tackiling hard deceptive problems with a new linkage learning genetic algortihm. In: Proc. of GECCO, AAAI (2002) 488–494
10. O'Neill, M., Brabazon, A.: mGGA: The meta-grammar genetic algorithm. In: Proc. of the 8th EuroGP. Volume 3447 of LNCS., Springer (2005) 311–320
11. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Proc. of the 3rd EuroGP. Volume 1802 of LNCS., Springer (2000) 121–132
12. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge, USA (1994)

Code Regulation in Open Ended Evolution

Lidia Yamamoto

Computer Science Department, University of Basel
Bernoullistrasse 16, CH-4056 Basel, Switzerland
Lidia.Yamamoto@unibas.ch

Abstract. We explore a homeostatic approach to program execution in computer systems: the "concentration" of computation services is regulated according to their fitness. The goal is to obtain a self-healing effect so that the system can resist harmful mutations that could happen during on-line evolution. We present a model in which alternative program variants are stored in a repository representing the organism's "genotype". Positive feedback signals allow code in the repository to be *expressed* (in analogy to gene expression in biology), meaning that it is injected into a reaction vessel (execution environment) where it is executed and evaluated. Since execution is equivalent to a chemical reaction, the program is consumed in the process, therefore needs more feedback in order to be re-expressed. This leads to services that constantly regulate themselves to a stable condition given by the fitness feedback received from the users or the environment. We present initial experiments using this model, implemented using a chemical computing language.

1 Introduction

In a world where computing devices are becoming ubiquitous, it is highly desirable to have systems that are able to operate continuously, adapting to the environment and tracking changing goals. Ideally, such constant self-optimisation should occur not only via hard-wired adaptive algorithms but also through evolution of new functionality without human intervention.

With open ended evolution the system evolves in the environment where it is used. Its fitness must be evaluated online during operation, as opposed to a a well-protected laboratory setting as the one used in off-line evolution. It is therefore particularly important to minimise the impact of potentially harmful genetic operations (mutation, crossover) on the running system. Classical computation models are ill suited for this problem: the typical sequential execution model leads to brittle programs in which adding, deleting or modifying a single instruction may have catastrophic effects.

We investigate the potential of artificial chemical computing models [1,2,3,4] for open ended evolution. In these models, change is a rule rather than an exception. Their high parallelism and multiset support allow programs to be expressed in terms of transformations (chemical reactions) applied to their internal objects (data or the transformation rules themselves, in case of high-order

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 271–280, 2007.
© Springer-Verlag Berlin Heidelberg 2007

models [5]).Chemical models have been shown that are intrinsically robust to random execution order [6], instruction deletion [7], inherent noise [8,9], and so on. They have been pointed as suitable for new, autonomic systems able to run unsupervised [10].

In chemical computing, code and data objects are represented as chemical substances that may occur at given *concentrations* in a reaction vessel. The reaction vessel is often implemented as a *multiset*, where elements may occur more than once. The higher the concentration of a given element type, the higher the probability that it participates in a chemical reaction, i.e. that it gets executed (potentially in combination with other elements) to produce a result.

Several reaction vessels might be composed in a hierarchical or recursive way [1,11] and vessels may communicate with each other by exchanging substances among themselves [11]. Substances react with each other according to specified reaction rules, and the result of the reaction goes to the multiset as well. Substances may be consumed in the reaction, or may act as enzymes for reactions without being consumed in the process.

Computations proceed as chemical reactions, leading to new substances that in turn react with others, and so on, potentially forming large reaction networks. These networks must be regulated to produce the desired result, and have been shown exhibit evolutionary ability [12].

When evolving computational chemical reaction networks online in an open ended context, it is important to ensure that the resulting program can be safely executed, or that any undesirable effects of the execution can be reverted. For this purpose we can take inspiration from biological evolution, which has endowed organisms with several protection mechanisms that ensure viable offspring with high probability. These include redundancy mechanisms such as redundant DNA to protein encoding, diploidy, gene duplication; and self-healing mechanisms such as the regulatory DNA repair cycle, in which proteins respond to DNA damage signals by triggering DNA repair mechanisms, or apoptosis (programmed cell death) when DNA repair is not possible.

We are currently exploring program regulation mechanisms, in which the concentrations of computation objects (at several levels of granularity, such as instructions, modules, programs, or higher level services) are regulated to produce a self-healing effect that resists harmful mutations that could happen during on-line evolution. Fitness evaluation and selection of suitable organisms form integral parts of such regulation cycles. Alternative program variants are stored in a repository which represents the organism's genotype. Code in the repository is *expressed* (in analogy to gene expression in biology) at regular intervals, meaning that it is injected into an execution environment where it is executed and evaluated. Since execution is equivalent to a chemical reaction, the program is consumed in the process.

Code expression happens in response to activation signals coming from the execution environment. These signals are generated in response to *fitness reward* signals coming from the applications evaluating the execution of the program with respect to its capacity to provide the intended service. On the other hand,

programs that are not suitable are evaluated with a low fitness, leading to a *punish* signal that travels to the repository to activate an inhibition rule which prevents the corresponding code to be re-expressed.

With such a mechanism, programs that have a higher fitness are expressed more often, increasing their concentration in the multiset, and therefore end up with a higher probability of being chosen for running.

Contrary to a classical approach where the presence or absence of a program, instruction or file is a binary variable, in such a chemical model objects are present at given concentrations which might increase or decrease in time. When the concentration reaches zero the object disappears from the system. This abstraction lends itself to *homeostatic* computer systems that constantly regulate themselves to a stable condition given by the fitness feedback received from the users or the environment. These systems would be able to adjust to changing requirements or conditions represented by a change in fitness, since they would be constantly seeking a fitness reward to survive.

This paper is structured as follows: Section 2 reviews the current state of the art, and section 3 presents our code regulation approach, with report on experiments in section 4.

2 State of the Art and Related Work

The term *chemical computing* refers to two distinct areas [1,3]: natural and artificial chemical computing. The first one uses real molecules and chemistry knowledge to build computational devices, e.g. in molecular/DNA computing. The second one derives computation models inspired by chemistry, which nevertheless run on traditional computers. This paper relates to the second area only. In this section we give a brief overview of some of the main artificial chemical computing models and their relation to evolutionary computing.

Artificial chemical computing models have been applied to diverse fields ranging from basic algorithms, to image processing applications, operating systems, compilers, dynamic software reconfiguration, [13], multi-agent systems, distributed computing, and, more recently, robotics [8], grid computing [14], and autonomic computing [15].

In *Gamma* [5] computations are modelled as interactions among atomic values that “float” freely in a chemical solution. These values are represented as elements in a *multiset*, an unordered set within which elements may occur more than once. The number of occurrences of a given element within the multiset is called the *multiplicity* of the element. The multiset contains the data to be processed as well as reaction rules of the form condition-action. Computations replace elements satisfying the condition by those specified in the action. A computation terminates when no more chemical reactions can take place. More recently, γ-calculus has been introduced as an extension of the original Gamma model to a higher-order calculus [5] in which rules are part of the multiset, such that they can also be transformed.

In *Membrane Computing* [1,11] computations (chemical reactions among *objects*) occur inside a cell-like *membrane structure*. Membranes can be recursively nested. As in Gamma, objects are represented as elements of a multiset. They can be transformed into other objects and can cross membranes.

In *Artificial Chemistries* [2] computations are modelled as chemical reaction networks which can be represented as bi-partite graphs with substrates and reaction rules as nodes. Contrary to Gamma and Membrane Computing, Artificial Chemistries do not focus on specific programming aspects, but rather on the emergent large-scale effects of the interactions among network elements. Chemical organisation theory [16] is used to bridge the gap between the microscopic behaviour at the code (reaction rule) level and the macroscopic behaviour of the system as a whole.

Artificial Regulatory Networks have been shown to model the biological regulatory mechanisms in both natural [9] and artificial systems [12]. In [12] a regulatory network is represented with a genotype/phenotype binary encoding in which genes express proteins, which in turn control the expression of genes, unleashing large reaction networks that evolve by gene duplication and mutations. These networks are able to compute functions, such as a sigmoid and a decaying exponential.

An *Algorithmic Chemistry* [6,4] is a reaction vessel in which instructions are executed in random order. In [6] the power of genetic programming (GP) applied to an algorithmic chemistry on evolving solutions to specific problems is shown. The authors point out the importance of the concentration of instructions, rather than their sequence. They start from a nearly unpredictable system in which execution of instructions at a random order leads to a random program output. This system is set to evolve by GP, including crossover and mutation of instructions placed in registers, obtaining at the end a highly reproducible output in spite of the random execution order.

Further examples of evolution using artificial chemical systems can be cited, such as the evolution of a robotic control system by GP [8], study of the emergence of evolution using organisation theory [17], evolution of artificial biochemical signalling networks able to compute functions [18], and so on. While these approaches show that chemical and evolutionary systems reinforce each other's potentials, the problem of open-ended general computing in these systems remains a challenging one.

3 Code Regulation

We have developed a code regulation system based on the fraglets chemical language [19]. An interpreter for this language is freely available for download at [20]. The language has a single structure, called a "fraglet" or "computation fragment". A fraglet is a string of symbols $[s_1 : s_2 : \ldots : s_n]k$ that may encode data, reaction rules involving two fraglets, or transformations of a single

fraglet. A suffix counter k indicates the multiplicity of the fraglet in the reaction vessel.

The fraglet language has been designed for network protocols. For this purpose the rule processing engine is based on a "tag matching" system in which fraglets are processed according to their head symbol, which is consumed in the process. This is similar to the way protocol headers in an incoming data packet are processed, consumed and then moved to a higher layer of a protocol stack.

Details on the fraglet language and instruction set can be found in [19]. For our purpose, only one type of rule is important: the reaction rule, called a *match*, or a *matchp* in its persistent variant. A *match* rule has the form $[match : s : tail_1]$, and when meeting a fraglet of the form $[s : tail_2]$ this results in a chemical reaction with product $[tail_1 : tail_2]$, i.e. the two fraglets are concatenated after eliminating their matching heads. The persistent variant *matchp* works in the same way, except that the original *matchp* rule is not consumed during the reaction. Using these rules and other simple transformation rules that manipulate fraglet strings, it is shown that communication protocols and other programs can be implemented [7,19]. The common point among all the programs shown in [7,19] is that *matchp* rules are dominant in the code, which makes it rather static as opposed to a real dynamic chemical system in which transformations happen most of the time.

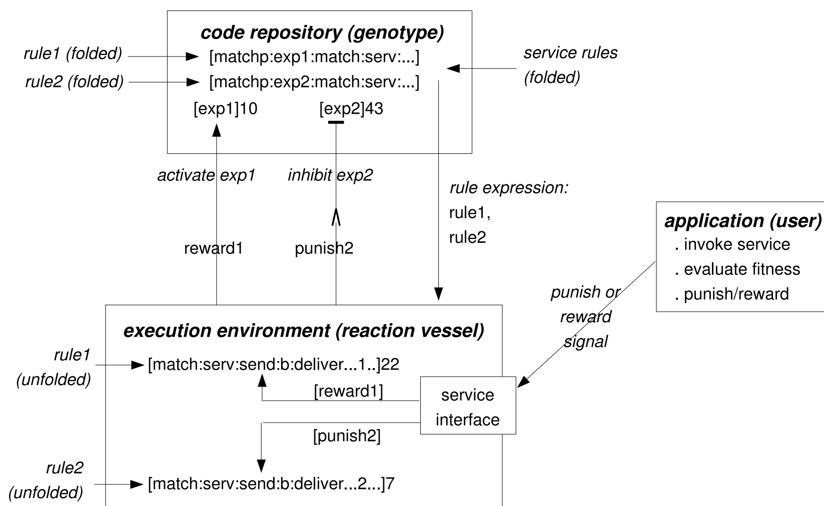


Fig. 1. Code Regulation Scheme

In contrast, we seek a solution in which the concentration of running code is controlled by activating/inhibiting factors and therefore not fixed in advance. We rely solely on the *match* rule and its multiplicity counter for this purpose. These rules are expressed at regular intervals based on the fitness feedback signals

received as a result on an execution that is assumed to provide a given service that can be evaluated during runtime.

Fig. 1 illustrates our code regulation scheme. Rules written as fraglets are stored in a code repository. The repository is also modelled as a reaction vessel, but counts on a persistent memory, so rules stored there are of the form *matchp*. Actually these rules are guarded by an exp_n rule expression factor, where n is the rule number. Rules in this form are said to be "folded". They can only be activated (expressed) by the proper $[exp_n]$ signal, also in the form of a fraglet. The existence of $[exp_n]$ in the repository triggers the corresponding $[matchp : exp_n]$ reaction that consumes one unit of $[exp_n]$ and produces a copy of the rule (free from the $[exp_n]$ guard, or "unfolded") which is injected into the reaction vessel. The rule has the form $[match : serv]$ where *serv* is the tag corresponding to the service that the rule implements. Examples of services could be to compute a function, or to transmit some data reliably to a destination.

The control mechanism consists then in regulating the amount ("concentration") of $[exp_n]$ in the repository according to the reward and punish signals received during the execution of the corresponding rule. Once inside the execution environment, the rule is activated when the service it performs is invoked by an application by injecting a request of the form $[serv : \ldots parameters \ldots]$. Note that the matching tag (*serv* in the example of Fig. 1) for both rules is identical, since they are alternative implementations of the same service. These alternative implementations compete for client invocations in the form of fraglets with the matching header tag $[serv : \ldots]$. The proportion of competing $[match : serv : \ldots]$ rules, given by their multiplicity counter, determines the probability of the match reaction to occur involving that rule.

Each rule executed is then either rewarded or punished according to the fitness feedback received from the application. Note that the application is unaware of which of the alternative rules executed the service; it sees only the service as a whole. A service interface is then in charge of translating these generic punish or reward signals to specific signals for the rule that actually performed the service. When rule n is punished, an inhibition signal in the form of a $[punish_n]$ fraglet is issued to the repository, which translates it to a $[match : exp_n]$ fraglet; the later cancels out one unit of exp_n expression signal for rule n. When rewarded, an activation signal $[reward_n]$ is issued for the rule, which translates directly to one or more $[exp_n]$ guards, that will then allow the rule to be re-expressed.

This cycle may continue indefinitely. Note however that the reward/punish signals are only injected by the application, so if there is no demand for a given service, the concentration of its rules will not change in steady state. A certain initial concentration of $[exp_n]$ may be present at the repository, which may cause a few rules to be expressed without demand at the beginning, in order to kick off the cycle; after that, the cycle is completely regulated by the feedback from the application. Since an unfolded rule has the form $[match : serv]$, it is consumed after providing the service, so the only chance for it to keep being executed regularly is to provide a good service that will be properly rewarded.

4 Experiments

We have performed some simulation experiments for code regulation on the fraglets platform. We used the CDP (Confirmed Delivery Protocol) implementation provided in the fraglets package [20] and enhanced it with the regulatory mechanism explained in the previous section.

CDP is a very elementary transport protocol that gets a payload from the application and transmits it to a given destination node in the network. We simulate then a network with two nodes, where the application consists of a data source (co-located with the code repository) and a data sink. Fitness evaluation is performed by the data source: it issues a reward unit upon confirmation of correct delivery of its payload, and a punishment unit when an error occurs or when no confirmation is received.

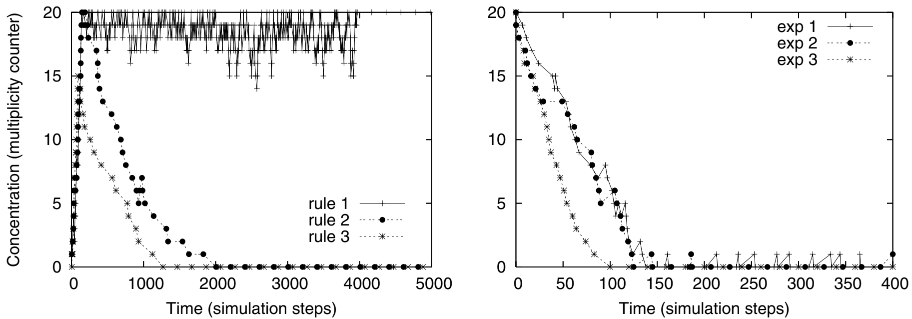


Fig. 2. Concentration of rules expressed to the reaction vessel (left) versus concentration of the corresponding expression signals (right)

Two experiments are shown. In each of them, three rules compete to provide the CDP service to the user: $rule_1$ is an always-good rule (i.e. transmits all the packets faithfully), $rule_3$ is always bad (never transmits anything), and $rule_2$ lies in between: it has a 50% probability of successful transmission.

In the first experiment, each reward for correct execution maps to an increase of a single unit of $[exp_n]$ in the code repository. Conversely, each punishment causes the decrease of one $[exp_n]$ unit, because a $[punish_n]$ fraglet translates into a $[match : exp_n]$ fraglet which eats up one unit of $[exp_n]$. If a rule is good all the time, it will only receive rewards, so each fired rule will lead to an extra unit of $[exp_n]$ being produced, which allows the rule to be re-expressed. On the other hand, expressing a rule consumes one unit of $[exp_n]$, so the concentration of a good rule does not increase in time. So in this experiment, good behaviour is just a way to stay alive in the system, one does not really get rewarded for it. Now if a rule is always bad, then it is punished with the decrease of one $[exp_n]$; on top of the one that had been used for its expression, it is then expected that the concentration of bad rules will rapidly decrease with successive punishments.

Fig. 2 shows the results of the first experiment. The repository is initialised with 20 $[exp_n]$ signals for each rule. The left side shows that the first rule reaches a fixed average concentration (with some oscillation due to the dynamic flow of signals) after a startup period. This concentration is equivalent to the amount of $[exp_1]$ signals that were initially present in the repository. The concentration of the third rule rapidly decreases as the reservoir of initial $[exp_3]$ gets depleted. The second rule lies in between, as expected. The relation between $[exp_n]$ control signals and expressed rules can be seen by comparing both plots on Fig. 2. For readability, the right side shows the concentration of $[exp_n]$ signals for the initial part of the simulation only, since during the remaining simulation time these concentrations do not significantly change.

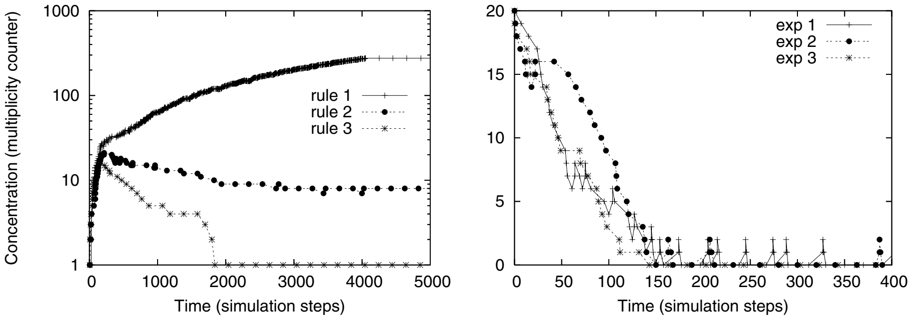


Fig. 3. Concentration of rules and activation signals, second experiment

In the second experiment, every reward for a well executed service triggers two activation signals $[exp_n]$, so the rule is re-expressed (to compensate for its consumption during execution) and at the same time its concentration is allowed to increase (due to an excess of activation signals coming from the rewards). This experiment simulates the situation in which the system provides an incentive for good services to proliferate in the system, and at the same time is able to kill bad ones. Fig. 3 shows the results of the experiment. We can now see that the concentration of $rule_1$ increases as long as it remains in use. Around $t = 4000$ simulation steps the rule is invoked for the last time, so the concentration of $rule_1$ stops growing. Rule 3 is eliminated early as before, while rule 2 now benefits from the extra reward to stay longer in the system, so after the service demand stops there are still around ten rules of type $rule_2$ in the vessel, which remain there in the absence of further control signals.

4.1 Discussion

Note that rules might not be discarded (reach a zero concentration) even if their behaviour is not ideal. This happens when feedback ceases, as with $rule_2$ in the second experiment. This is not necessarily bad, since these “dormant” rules might represent a useful source of genetic variability which may be helpful for

evolution. When conditions change, these rules might be reactivated and become useful in another context, where different service requirements or different environment conditions lead to different performance and resulting fitness.

Currently there is no mechanism to "destroy" rules once they are expressed. The only way to consume a rule is to fire (use) it. It is therefore difficult to punish programs not for low performance, but for lack of demand. One possibility would be to introduce an "aging" mechanism, so that rules that are not invoked for a long time "decay". This would be a useful mechanism to deprecate obsolete software. However, one concern is how to regulate this ageing mechanism itself: how slow or how fast should a service age? if it ages too quickly, the service becomes prematurely unavailable, whereas if it ages too slowly the system gets polluted by obsolete pieces of software (note that this is the situation today, as this "aging" process is controlled by humans in an ad hoc fashion).

The experiments shown are very simple, and intended to shed light on the chemical programming mechanism for code regulation. The next step is to extend them to complex regulation networks in which services are composed of numerous interconnected modules, each of which must be evaluated and selected. For this purpose we consider applying concepts from artificial regulatory networks [9,12] and chemical organisation theory [16] to fraglet code regulation.

5 Conclusions and Outlook

We have shown a regulation mechanism intended to provide resilience to open ended evolution. This is only one building block for such evolutionary process. The next step is then to show how evolution would actually happen in this context. We believe that the problem of discontinuity of the search space in genetic programming (a small change in a program may cause a big leap in fitness) could be minimised by allowing program or instruction concentrations to be expressed and controlled, instead of binary presence/absence of a given program. This may lead to more stable and predictable transformations in which a small change in a service implementation (which is made of variable concentrations of competing varieties) could lead to a comparably small leap in fitness from the point of view of the user. This cannot be easily achieved with conventional programming languages in which changing a single line of code in an otherwise perfectly working program may cause it to crash, loop or destroy resources. Chemical systems with regulated computations could lead to a much more elastic way to handle exceptions and therefore to a robust way to support online evolutionary computation for systems that may be made to run forever uninterrupted.

References

1. Calude, C.S., Paun, G.: Computing with Cells and Atoms: An Introduction to Quantum, DNA and Membrane Computing. Taylor & Francis (2001)
2. Dittrich, P., Ziegler, J., Banzhaf, W.: Artificial Chemistries – A Review. Artificial Life **7**(3) (2001) 225–275

3. Dittrich, P.: Chemical Computing. In: Unconventional Programming Paradigms (UPP 2004), Springer LNCS 3566. (2005) 19–32
4. W.G.Lasarczyk, C., Banzhaf, W.: An Algorithmic Chemistry for Genetic Programming. In: Proc. 8th European Conference on Genetic Programming. M. Keijzer, A. Tettamanzi, P. Collet, M. Tomassini, J. van Hemert (Eds.) Springer LNCS 3447, Lausanne, Switzerland (2005) 1–129
5. Banâtre, J.P., Fradet, P., Radenac, Y.: A Generalized Higher-Order Chemical Computation Model with Infinite and Hybrid Multisets. In: 1st International Workshop on New Developments in Computational Models (DCM'05). (2005) 5–14 To appear in ENTCS (Elsevier).
6. Banzhaf, W., Lasarczyk, C.: Genetic Programming of an Algorithmic Chemistry. In: Genetic Programming Theory and Practice II, O'Reilly et al. (Eds.). Volume 8. Kluwer/Springer (2004) 175–190
7. Tschudin, C., Yamamoto, L.: A Metabolic Approach to Protocol Resilience. In: Proc. 1st Workshop on Autonomic Communication (WAC). Springer LNCS 3457, Berlin, Germany (2004) 190–205
8. Ziegler, J., Banzhaf, W.: Evolving Control Metabolisms for a Robot. Artificial Life **7**(2) (2001) 171–190
9. Leier, A., Kuo, P.D., Banzhaf, W., Burrage, K.: Evolving Noisy Oscillatory Dynamics in Genetic Regulatory Networks. In: Proc. 9th European Conference on Genetic Programming. Springer LNCS 3905, Budapest, Hungary (2006) 290–299
10. Banâtre, J.P., Radenac, Y., Fradet, P.: Chemical Specification of Autonomic Systems. In: Proc 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04). (2004) 72–79
11. Paun, G.: Computing with Membranes. Journal of Computer and System Sciences **61**(1) (2000) 108–143
12. Kuo, P., Banzhaf, W., Leier, A.: Network topology and the evolution of dynamics in an artificial genetic regulatory network model created by whole genome duplication and divergence. Biosystems **85** (2006) 177–200
13. Wermelinger, M.A.: Specification of Software Architecture Reconfiguration. PhD dissertation, Universidade Nova de Lisboa, Lisbon, Portugal (1999)
14. Banâtre, J.P., Fradet, P., Radenac, Y.: Towards Grid Chemical Coordination. In: Proceedings of Symposium on Applied Computing (SAC). (2006) (short paper).
15. Banâtre, J.P., Radenac, Y., Fradet, P.: Chemical specification of autonomic systems. In: Proc 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04). (2004) 72–79
16. Matsumaru, N., Centler, F., di Fenizio, P.S., Dittrich, P.: Chemical Organization Theory as a Theoretical Base for Chemical Computing. International Journal of Unconventional Computing (in print) (2006)
17. Matsumaru, N., di Fenizio, P.S., Centler, F., Dittrich, P.: On the Evolution of Chemical Organizations. In: Proc. 7th German Workshop on Artificial Life. (2006) 135–146
18. Deckard, A., Sauro, H.M.: Preliminary Studies on the In Silico Evolution of Biochemical Networks. ChemBioChem **5**(10) (2004) 1423–1431
19. Yamamoto, L., Tschudin, C.: Experiments on the Automatic Evolution of Protocols using Genetic Programming. In: Proc. 2nd Workshop on Autonomic Communication (WAC), Athens, Greece (2005) 13–28
20. Fraglets Home Page: http://www.fraglets.net/ (2005)

Data Mining of Genetic Programming Run Logs

Vic Ciesielski and Xiang Li

RMIT University, Melbourne, Vic 3001, Australia
vc@cs.rmit.edu.au
http://www.cs.rmit.edu.au/∼vc

Abstract. We have applied a range of data mining techniques to a data base of log file records created from genetic programming runs on twelve different problems. We have looked for unexpected patterns, or golden nuggets in the data. Six were found. The main discoveries were a surprising amount of evaluation of duplicate programs across the twelve problems and one case of pathological behaviour which suggested a review of the genetic programming configuration. For problems with expensive fitness evaluation, the results suggest that there would be considerable speedup by caching evolved programs and fitness values. A data mining analysis performed routinely in a GP application could identify problems early and lead to more effective genetic programming applications.

1 Introduction

Using genetic programming to solve a problem is a complex task. There are many aspects of the configuration that need to be determined and the various components and parameter values interact with each other in ways that are not obvious. Typically plots of fitness vs generations are used to monitor the evolutionary process and it is considered that all is well as long as fitness keeps improving with generations. However, it is possible that the development process could be improved if there was more insight into the interactions between components and parameters. We suggest that some of this insight can be gained by using techniques from data mining to analyse log files from genetic programming runs.

Data mining [1] is a well established approach for extracting useful information from large volumes of data. Since many run logs comprise a large volume of data it is reasonable to ask whether data mining techniques can be used to find useful information in this large volume of data. One metaphor for data mining is a search for “golden nuggets”. A golden nugget is an interesting, unexpected fact or relationship that was previously unknown. We want to see if we can find any golden nuggets in the run log data.

Data mining is relatively mature and there are number of systems available. We have used one of these, the WEKA system [1]. Weka provides a large number of algorithms for the data mining tasks of classification, clustering, attribute selection and visualisation.

Related work: Our work is related to previous work on visualisation in evolutionary computing and fitness landscape analysis in genetic programming. In the

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 281–290, 2007.
© Springer-Verlag Berlin Heidelberg 2007

visualisation area, Collins [2] describes an approach to visualising the progress of a genetic algorithm run at a number of levels of abstraction using specialised visualisation software. Daida et al. [3] describe a method of visualising the set of tree shapes that emerge during an evolutionary run. Ekart and Gustafson [4] present alternative visualisations of tree shapes that emerge during a run and show that, for their symbolic regression problem, there are differences between the tree shapes of the best and worst programs. In the fitness landscape area, early work by Kinnear [5] analysed the fitness landscapes of a number of simple problems and correlated problem difficulty with a number of properties of the landscapes. In [6] Langdon and Poli have analysed the Santa Fe ant problem in detail and have given 3D visualisations of a number of landscape properties. Vanneschi et al [7] have looked at neutrality, i.e, areas of the fitness landscape that have the same fitness and shown a number of visualisations.

Most of the previous work tends to focus on one or two problems and uses specialised visualisation programs. Our approach is quite different in flavour. We have generated log files of the genetic programming runs for many problems and performed a data mining exercise, using a standard package, on the these log files. We have looked for any significant recurring or unusual patterns in the run logs. Our motivation is that if we can find significant problem patterns we can perhaps take steps to solve the problems and improve the practice of genetic programming.

1.1 Research Questions

Our overall aim is to determine whether any significant patterns, or golden nuggets, can be found in a data warehouse of genetic programming run logs. In particular we are interested in:

1. Do the data mining techniques of visualization, classification, clustering and association finding reveal any interesting patterns or golden nuggets.
2. Are there any patterns in the shape, size, depth and fitness of the program trees generated?
3. How many duplicate individuals are evaluated in a run?
4. Are any improvements to the GP process suggested by the discovered patterns?

2 The Data

We have collected data for a number of toy problems and real world problems. The toy problems are used extensively in genetic programming research and are listed in the top half of table 1. For the toy problems, that is, Santa Fe Ant[6], Max[8], Lawnmower[9], Symbolic Regression[9] and 5Parity[9] we performed runs using genetic algorithm parameters from the literature. The real world cases in the bottom half of table 1 represent significant effort on using genetic programming to solve difficult and very difficult problems. For these problems we asked our colleagues to log their next 5 runs and give us the log files.

For this paper we have extracted the following data for each of the problems.

1. **Run Number.** There are 5 runs.
2. **Generation number.** Maximum number of generations is 100.
3. **Fitness.** This is a numeric value where 0 indicates the best fitness. Depending on the problem this could take a small number of integer values or a large range of real values.
4. **Program size.** This is the number of nodes in the program tree.
5. **Tree depth.**
6. **Program.** This is the full text of the evolved program. The following is an example program from a run with the Santa Fe ant problem:

```
(Prog2 (IfFoodAhead (Prog3 (Prog3 turnRight turnRight move)
(IfFoodAhead turnLeft turnLeft) (IfFoodAhead turnRight move))
(Prog3 turnRight move turnLeft)) (IfFoodAhead (Prog2 move
turnLeft)  (Prog2 turnLeft turnRight)))
```

7. **Tree Shape.** This is the the tree shape corresponding to the program. The tree shape is obtained by removing all function names from the program text and replacing all terminal symbols with `#`. The tree shape of the above program is:

```
((((###)(##)(##))(###))((##)(##)))
```

3 Methodology

From the data preparation step we have 12 files, each containing the seven fields, or attributes in data mining terminology, described in the previous section. Each record in a file represents the evaluation of a candidate individual. The data are from 5 independent runs and the number of records in each file is given by the "Total Eval" column in table 1.

To perform the data mining functions we have used the WEKA machine learning system [1]. In some cases a sample of 10,000 was extracted at random because of memory and computational constraints. Discretization of the numeric variables was performed as needed.

We began with visualizations of the data and then moved on to the classification, clustering and association finding functions available in the WEKA system. The visualizations suggested further specific analysis of duplicate individuals which was achieved by separate scripts and programs.

3.1 Visualizations

The WEKA system provides two main ways of visualizing the data: (1) The distribution of values of a single attribute can be visualized as in figure 1, and (2) Three attributes can be visualized together, one on the x-axis, one on the y-axis and one as the colour, as in figure 2. In this mode a thumbnail panel of plots for each pair of attributes makes it quite easy to visually identify interesting looking patterns for further investigation. We have done these visualizations for each of the 12 problems.

3.2 Redundant Computation

The visualizations revealed that a significant number of programs were being evaluated several times and we decided to look further into this. Redundant computation occurs when a program whose fitness was evaluated in an earlier generation is re-generated in a later generation or a logically equivalent program is generated. This new program must now be re-evaluated, but the fitness will be the same as before so this computation can be regarded as redundant or wasted. Two programs are logically equivalent if they perform the same underlying computation, for example the programs `(+ A B)` and `(IF (= 1 1) THEN (+ A B) ELSE C)`, as are `(+ A 1)` and `(+ A (/ A A))`. In general determining whether two programs are logically equivalent undecidable. However, in some cases, programs which have the same shape and the same fitness ($SSSF$), but are not string equivalent (SE), will be logically equivalent. In symbolic regression and classification problems which use just the function set {+,-,*,/}, for example, {+,*} are commutative. In this case, if two programs have the same shape and fitness, but are not string identical, it is fairly certain that some of the same operations are being done in a different order, for example `(+ A B)` and `(+ B A)`. Thus $SSSF - SE$ evaluations are most likely of logically equivalent programs. For problems like the Santa Fe ant, where the operators are not commutative inferences about logically equivalent programs are not possible, but we have shown data in table 1 for completeness.

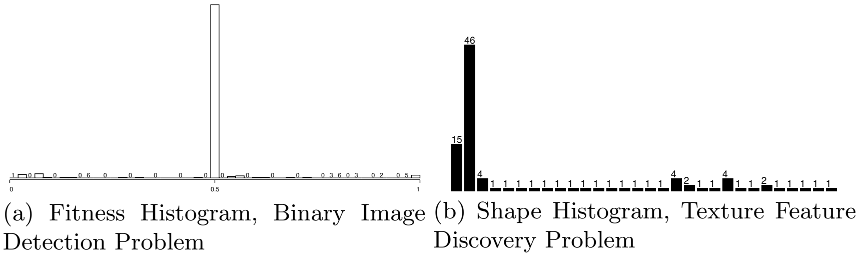
3.3 Fraction of Possible Tree Shapes Explored

After looking at the visualizations of the tree shapes and noting the large numbers of different shapes generated in the various runs, we became interested in determining how many of the possible shapes were being explored. Since most of the problems use binary trees we have used results from [10,11] which give the total number of different binary trees for a given depth $d, d = 1, 2, 3...9$. This sequence is $1, 3, 21, 651, 457653, 2.10E + 11, 4.4E22, 1.9E + 45, 3.7E60$. This is a 'double exponential' [11] sequence and after a depth of 5 the number of possible trees becomes massive. The Santa Fe ant problems have one ternary function. For this problem we have used a depth of 5. The number of possible tree shapes is thus the number of trees with nodes of degree 3 or 4, for depths up to and including 5. We have roughly estimated this number as follows: There are $4.5E5$ binary trees of depth 5. There are at least this number of ternary trees. So the number of mixed binary and ternary trees is at least $4.5E5 \times 4.5E5 = 2.0E11$. We have used the same approximation for the soccer problem which also has some ternary functions. The results of this analysis are presented in table 2.

4 Results

4.1 Visualizations

The visualisations of single attributes revealed two unexpected patterns. These are shown in figure 1.



(a) Fitness Histogram, Binary Image Detection Problem

(b) Shape Histogram, Texture Feature Discovery Problem

Fig. 1. Visualizations of Single Attributes

Golden Nugget 1: Unexpected fitness distribution in the binary image problem.

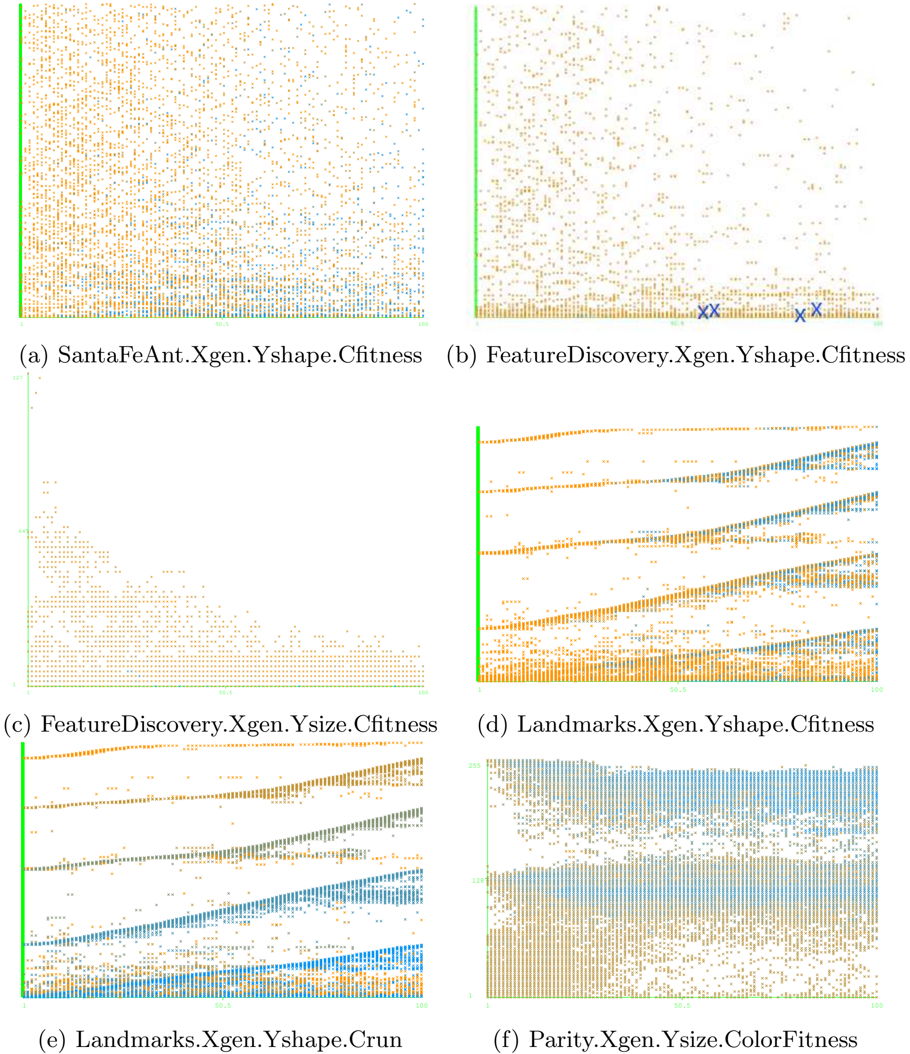
In visualizing fitness distributions of the various problems we found the histogram in figure 1a. This distribution of fitness was very unexpected. There is a large spike at a fitness of 0.5. The problem involves a binary image classification task where circles need to be distinguished from squares. Individual pixels and blocks of pixels have been manually removed from a solid black circle/square on a white background leaving irregular objects that a human would readily recognize as circles or squares, but which is a difficult machine learning problem. Inspection of the training data revealed that about half of the examples were quite straight forward and most programs got these quite easily leading to a local optimum.

There was a similar finding in the soccer problem, all fitness values were 0. This turned out to be because a "goals-only" fitness function was being used in the runs and no goals were scored in any of the runs. This finding was not a surprise to the collaborator who provided the runs.

Golden Nugget 2: Unexpected distribution of tree shapes in the feature discovery problem.

There were a large number of programs for tree shape '#' for the texture feature discovery problem. These are single node programs. The histogram is shown in figure 1b. The number of single node programs being evaluated is very high. This appears to be an unexpected consequence of using a very limited function set and a size penalty in fitness evaluation in order to get small, understandable programs.

After looking at the visualisations of the single attributes we then looked at the visualisations of combinations of 3 attributes. Some of these are shown in figure 2. The data have been randomized prior to visualization. The captions for each figure have the form *Problem.Xattribute.Yattribute.C(olour)attribute* and indicate the problem, the attribute shown on the x-axis, the attribute shown on the y-axis and the attribute rendered as the colour. Figure 2a shows generations, shape and fitness for the Santa Fe ant problem. For the fitness attribute, orange



(a) SantaFeAnt.Xgen.Yshape.Cfitness (b) FeatureDiscovery.Xgen.Yshape.Cfitness

(c) FeatureDiscovery.Xgen.Ysize.Cfitness (d) Landmarks.Xgen.Yshape.Cfitness

(e) Landmarks.Xgen.Yshape.Crun (f) Parity.Xgen.Ysize.ColorFitness

Fig. 2. Visualizations. Code: *Problem.Xattribute.Yattribute.C(olour)attribute*. Orange indicates low fitness, blue high fitness[1].

indicates low fitness and blue indicates high fitness. Shape is a non numeric variable and is treated as follows: If a shape occurs in the data it is assigned a row on the y axis. The order is random. If a particular shape occurs at generation x an `x` is printed at the corresponding point on the grid in a colour based on the fitness value. If there are several occurrences of a shape in that generation,

[1] Colour versions of these images can be found at www.cs.rmit.edu.au/∼vc/papers/eurogp07.pdf.

the colour is randomly selected. Figure 2a shows that, for this problem, a wide range of tree shapes is explored, that fitter shapes become more numerous as the generations increase and that highly fit programs are reasonably well distributed throughout the different shapes. This pattern is generally expected and was evident in a number of the problems.

Figure 2b shows a qualitatively different situation to figure 2a. Here, there are very few fit programs and they are tied to specific shapes. Figure 2b has been manually enhanced because the high fitness points were so infrequent that they disappeared when the image was reproduced in reduced size for publication.

Golden Nugget 3: Very few fit programs in the feature discovery search space.

In the feature discovery problem there are very few programs of high fitness and they are associated with a very small number of tree shapes. This finding is supported by figure 2c which shows program size rather than shape on the y axis.

Figure 2c shows that program size decreases with increasing generations. This is consistent with the size penalty in the fitness function that is being used in this problem.

Figure 2d shows a visualization that has been constructed slightly differently to the previous ones. Here the shapes are not allocated at random on the y-axis as before, but in the order that they were generated during the runs. The different runs are clearly evident in the 5 bands. The figure suggests that each run has mostly explored a new set of shapes. This is confirmed in figure 2e where the colour now denotes run number. The higher density of points at the bottom of figure 2e indicates shapes that have been explored in all runs. Figure 2f shows a bimodal distribution of fitness in the parity problem with high values clustered around sizes 110 and 225 suggesting a deceptive problem.

4.2 Analysis of Redundant Computation

The results of an analysis of the number of duplicate programs evaluated are shown in table 1. The table shows the number of string equivalent programs evaluated and the number of programs evaluated with the same shape and the same fitness. For the Santa Fe Ant problem, for example, a total of 211,500 fitness evaluations were performed. Of these 109,614 (52%) were of string-equivalent programs, that is 52% of the evaluations were expended on programs that had been evaluated at least once before. The 'MFE' (Most Frequently Evaluated) column gives the percentage of evaluations accounted for by the individual evaluated the most number times, in this case 0.6%. The right hand part of the table shows the same data for programs that have the same shape and the same fitness. The LR (Logically Redundant) column is only given cases where, based on domain knowledge, it is highly likely that programs with the same shape and

Table 1. Analysis of equivalent individuals over 5 runs. MFEI - Most Frequently Evaluated Individual, LR - Logically Redundant.

Problem	Total	String-Equivalent			Same Shape/Fitness			
	Eval	Number	Pcnt	MFEI	Number	Pcnt	LR	MFEI
Santa Fe Ant	211,500	109,614	52%	1.3%	163,596	77%		2.3%
Modif Santa Fe Ant	177,000	109,088	62%	2.6%	148,740	84%		2.7%
Max, depth 5	26,200	23,359	89%	22.1%	23,910	91%	2%	22.2%
Lawnmower	252,500	70,575	28%	0.6%	115,469	46%		1.0%
Symbolic Regression	252,500	54,720	22%	0.6%	71,467	28%	6%	0.6%
5Parity	252,500	41,719	16%	2.3%	150,964	60%	44%	2.3%
Binary Image1	46,800	4,565	10%	0.6%	34,506	74%		3.5%
Binary Image2	35,800	6,783	19%	1.2%	23,906	67%		2.6%
Goal Scoring in Soccer	50,500	16,857	33%	1.4%	42,905	85%		4.7%
Cephalometric Landmarks	50,500	14,728	29%	1.5%	33,665	67%	38%	12.2%
Texture Feature Discovery	50,200	39,210	78%	3.2%	39,622	79%	1%	3.2%
Texture Classification	125,000	75,910	61%	0.5%	106,977	85%	24%	19.9%

fitness are logically equivalent. Since elitism is being used, some degree of duplicate evaluation is expected. However, the overall amount of duplicate evaluation across all of the problems is surprisingly high and there is considerable variation in the amount of redundant evaluation across the different problems. Inspection of the table reveals:

Golden Nugget 4: There is a surprisingly high level of redundant computation in most of the problems.

Golden Nugget 5: There is a large variation in the amount of redundant computation across the problems

The max problem stands out on all measures of redundant computation. This is not surprising due to the idiosyncratic nature of the problem. There are only two functions and one terminal, and the solution is known to be a full tree.

Table 2 provides more details about the tree shapes generated in a randomly selected run. ME stands for Most Evaluations. For the max problem 274 distinct tree shapes were evaluated. This is 40.5% of the total number of binary trees possible at depth 5. Most of the evaluations (78%) were carried out on trees of depth 5, which is the maximum depth for this problem. Also, most of the evaluations were carried out on trees containing 23 (s23)nodes. The Santa Fe ant and ascii soccer problems, while they also have a maximum depth of 5, have a much larger number of possible shapes as there are some ternary functions. It is evident from the table that once the tree depth is larger than 5 or 6 only a very small fraction of the the possible tree shapes will be examined. For most of the problems, most of the computation is performed on trees of maximum depth, the last 3 problems are an exception, an unexpectedly large amount of computation is being performed on single node trees.

Table 2. Tree Shapes and Most Evaluations (ME), 1 Run. * indicates this is not Max Depth.

Problem	Distinct Shapes	% of Possible	ME at Depth	ME at Size
Santa Fe Ant	5,403	< 2.7E-6	82% (d5)	22% (s29)
Modif Santa Fe Ant	3,837	< 1.9E-6	78% (d5)	13% (s14)
Max	274	40.5%	78% (d5)	11% (s23)
Lawnmower	17,486	< 9.2E-10	18% (d7)	1% (s75)
Symbolic Regression	22,769	1.2E-41	95% (d8)	6% (s75)
5Parity	9,306	4.8E-42	88% (d8)	19% (s239)
Binary Image 1	2,661	6.0E-20	69% (d7)	11% (s29)
Binary Image 2	1,240	2.8E-08	73% (d7)	14% (s29)
Ascii Soccer	1,844	< 9.2E-7	66% (d5)	11% (s5)
Cephalometric Landmarks	2,432	1.2E-08	*13% (d6)	12% (s1)
Texture Feature Discovery	437	9.9E-21	*46% (d1)	46% (s1)
Texture Classification	3,036	$\ll$E-42	*36% (d1)	36% (s1)

The feature discovery problem stands out in table 1. Very few shapes are explored and the number of single node programs evaluated is very high. This, together with figure 2b, suggests:

Golden Nugget 6: The texture feature discovery problem has pathological behaviour.

5 Conclusions

Our primary aim was to determine whether any golden nuggets could be found in genetic programming run logs using data mining tools and methodologies. In this we have been successful, uncovering six previously unknown patterns that had a significant effect on the genetic programming solution to a problem. We performed a large number of experiments involving visualisation, classification, clustering and association finding. The most productive data mining technique was visualization. The other techniques did not reveal anything additional to the visualisations and we have not included the details of the runs and results in this paper. We found that in most of the problems each run explored a different set of tree shapes. We also found that there was a surprisingly high amount of evaluation of duplicate individuals.

In terms of improvements to the genetic programming process our results suggest three things: (1) For problems in which there is a high computational cost in fitness evaluation it is worth implementing a program cache. (2) Use of size penalties in fitness functions needs to be done with caution. If the penalty is too big, much computation might be spent in unproductive areas of the search space. (3) This kind of data mining analysis performed routinely in a genetic programming application could identify problems early and lead to more effective

genetic programming applications. The additional effort is not high because the data mining tools now available provide very good visualization and classification tools.

Acknowledgments. We would like to thank Andrew Innes, Brian Lam, Andy Song, Andrei Bajurnow and Teja Nanduri for providing us with log files of their runs. Part of this work was supported by grant EPPNRM054 from the Victorian Partnership for Advanced Computing.

References

1. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. 2nd edn. Morgan Kaufmann, San Francisco, California (2005)
2. Collins, T.D.: Visualizing evolutionary computation. In: Advances in evolutionary computing: Theory and applications. Springer-Verlag New York, Inc., New York, NY, USA (2003) 95–116
3. Daida, J.M., Hilss, A.M., Ward, D.J., Long, S.L.: Visualizing tree structures in genetic programming. Genetic Programming and Evolvable Machines **6**(1) (2005) 79–110
4. Ekart, A., Gustafson, S.: A data structure for improved GP analysis via efficient computation and visualisation of population measures. In Keijzer, M., O'Reilly, U.M., Lucas, S.M., Costa, E., Soule, T., eds.: Genetic Programming 7th European Conference, EuroGP 2004, Proceedings. Volume 3003 of LNCS., Coimbra, Portugal, Springer-Verlag (2004) 35–46
5. Kinnear, Jr., K.E.: Fitness landscapes and difficulty in genetic programming. In: Proceedings of the 1994 IEEE World Conference on Computational Intelligence. Volume 1., Orlando, Florida, USA, IEEE Press (1994) 142–147
6. Langdon, W.B., Poli, R.: Why ants are hard. In Koza, J.R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M.H., Goldberg, D.E., Iba, H., Riolo, R., eds.: Genetic Programming 1998: Proceedings of the Third Annual Conference, University of Wisconsin, Madison, Wisconsin, USA, Morgan Kaufmann (1998) 193–201
7. Vanneschi, L., Pirola, Y., Collard, P.: A quantitative study of neutrality in gp boolean landscapes. In: GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation, New York, NY, USA, ACM Press (2006) 895–902
8. Langdon, W.B., Poli, R.: An analysis of the MAX problem in genetic programming. In Koza, J.R., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M., Iba, H., Riolo, R.L., eds.: Genetic Programming 1997: Proceedings of the Second Annual Conference, Stanford University, CA, USA, Morgan Kaufmann (1997) 222–230
9. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)
10. Sloane, N.J.: Number of Binary Trees of Height n, On-Line Encyclopedia of Integer Sequences. http://www.research.att.com/cgi-bin/access.cgi /as/njas/sequences/eisA.cgi?Anum=A001699 (Accessed 11-Oct-2006)
11. Aho, A., Sloane, N.: Some doubly exponential sequences. Fib. Quart **11** (1973) http://www.research.att.com/∼njas/doc/doubly.html.

Evolving a Statistics Class Using Object Oriented Evolutionary Programming

Alexandros Agapitos and Simon M. Lucas

Department of Computer Science
University of Essex, Colchester CO4 3SQ, UK
aagapi@essex.ac.uk, sml@essex.ac.uk

Abstract. Object Oriented Evolutionary Programming is used to evolve programs that calculate some statistical measures on a set of numbers. We compared this technique with a more standard functional representation. We also studied the effects of scalar and Pareto-based multi-objective fitness functions to the induction of multi-task programs. We found that the induction of a program residing in an OO representation space is more efficient, yielding less fitness evaluations, and that scalar fitness performed better than Pareto-based fitness in this problem domain.

1 Introduction

The majority of programs currently being developed are written in Object-Oriented languages such as Java, C++, C# and Smalltalk. The OO paradigm provides an ingenious, general-purpose conceptual framework for the software industry to engineer scalable, manageable software. The four major elements of this model are: *Abstraction*, *Encapsulation*, *Modularity*, and *Hierarchy* [1]. This conceptual framework and the technology that it encompasses, provides an excellent software development space for human programmers to design and implement solutions to complex problems.

The vast majority of evolved programs use a functional expression tree representation, and while GP has produced some impressive results, it has significant problems with scalability. Most GP evolved programs are simple expression trees with constant time complexity, rather than being general programs. Current GP ignores much of what we know about how to design well structured software, which to a significant practical degree, means object oriented software. To quote Langdon [2]: "Genetic programming, with its undirected random program creation, would appear to be the anathema of highly organised software engineering".

In this paper we propose a hypothesis on the efficiency of evolving programs specified in an OO programming space. We show that for a particular type of problem, Object classes with cooperating member methods that inspect and modify the object's internal state provide a more appropriate unit of evolution than the essentially unstructured Koza's *ADF* approach to modular program representation. Of direct relevance to this work is the work of Langdon [2] on

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 291–300, 2007.
© Springer-Verlag Berlin Heidelberg 2007

evolving abstract data types and of Bruce [3] on Object Oriented Genetic Programming. Langdon and Bruce independently evolved data types such as stacks, queues, priority queues, and linked lists.

As a motivating example, we tackle the problem of evolving a program to calculate some statistical measures on a set of numbers. We compare the efficiency of evolving such a program by allowing the Evolutionary Algorithm (EA) to operate on two different representation spaces, namely, OO and functional, and for each method compare performance when using *scalar* and *Pareto-based* fitness functions.

2 Object Oriented Versus Functional Program Spaces

The *Statistics* program is required to exhibit functionality for querying the number of values in the statistical sample, calculating the mean, variance, and standard deviation of the sample. As with most programming problems, there are many possible implementations and we can encourage the EA to induce a specific implementation by allowing it to work on a particular programming space. The interfaces presented in figure 1 show the signatures of the operations that provide the desirable functionality to model the statistics of a fix of numbers. However, the implementations of the `OOStatistics` and `FunctionalStatistics` interfaces reside in the OO and functional programming spaces respectively.

In the class that implements `OOStatistics` it is not necessary to store all the numbers of the sample; it is sufficient to keep a running total of how many, their sum and their sum of squares. Observing the methods of `OOStatistics` we note that an additional `addSample` method is declared to allow the update of instance variables each time a new value is added to the sample. This is indeed a crucial characteristic of Class objects, the notion of *object state*, which encompasses collectively all the properties of the object along with the current values of each of these properties. In the case of the *Statistics* class, the object state consists of three instance variables, namely, n (the number of values added), *sum* (the sum of values), and *sum_square* (the sum of squares of values).

On the other hand, the signatures of the operations composing a functional, Koza-style, modular program for performing statistics on a sample of values are presented in the interface `FunctionalStatistics`. Using Koza's ADF terminology, the statistics program has four *result producing branches* that allow further hierarchical references among them. While there exist GP variants that have been operating in a procedural space, by providing some form of state manipulation via global variables, we intentionally study the pure functional arena that traditional GP has been widely applied. This space is defined as the set of all finite mappings from inputs to outputs in a particular problem domain. Here, we choose to use explicit recursion as a means to iterate over the elements of the input list passed as a parameter to each function. To avoid the problem of unending recursion we set the number of allowable recursive calls to be slightly bigger than the length of the input list.

```
public interface OOStatistics{
  public double addToSample(double d);
  public double n();
  public double mean();
  public double variance();
  public double stdDeviation();
}
```

```
public interface FunctionalStatistics{
  public double n(NList list);
  public double mean(NList list);
  public double variance(NList list);
  public double stdDeviation(NList list);
}
```

Fig. 1. The interfaces specifying the signatures of the evolvable methods under OO and functional representation spaces

3 Evolvable Class Representation

Following previous work on the evolution of multi-task programs [2,3] we decided to represent an evolvable individual using a multi-tree structure. For the sake of our discussion here we shall call this structure an *Evolvable Class*. The syntactic structure of an *Evolvable Class* couples a linear data structure of *class* and *instance variables* (representing the object state) along with a set of *evolvable methods* (using an expression tree representation) that are responsible for the way an object acts and reacts, in terms of state changes and message passing.

Traditionally, the use of memory within GP takes the form of either *scalar* or *indexed* memory [2]. Object state encompasses those properties that contribute to making an object uniquely that object. It was felt that the discrete nature of those properties can be better represented using scalar memory cells.

The evolutionary run initialization performs a random sampling of *Evolvable Class* structures. Each *Evolvable Class* contains a set of expression trees representing the methods declared in the `OOStatistics` interface. Their argument and return types are specified accordingly. These expression trees are generated using the ramped-half-and-half algorithm. Subsequently, a series of independent random choices, using a uniform probability distribution, is made for the number and type of instance variables that compose the object state. Here, we allow a maximum of ten instance variables. For instance variable types, it is reasonable to draw possible useful instances from the programming space under consideration. The yet-to-be-evolved program needs to operate on *numeric* data values. Thus, initially, we define `Double` as the sole type of instance variable.

4 Experimental Methodology

A series of experiments have been conducted to explore the issues of program representation offered by the OO programming paradigm. We use Koza's ADF approach as a benchmark to compare the efficiency of evolving target solutions specified in OO and functional program spaces respectively. In experiment series E_{OO} we evolve an object oriented statistics program that enjoys the co-operative application of instance methods that inspect and modify the object's memory. We investigate two different variations of object state organization. In experiment E_{OO_1} we use a preordained layout of memory, that is we apply our

knowledge of the problem domain to manually set the number of required state variables, in this case 3 (*n*, *sum*, and *sumSquares*. In experiment E_{OO_2} we allow the organization of object state variables to be emergent through an evolutionary fitness-driven process. That is, during the evolutionary run initialization we perform a random sampling of program and object state spaces, discussed in section 3, causing certain suitably configured *Evolvable Classes* to prosper in later generations of the population.

Table 1. Primitive elements for evolving a statistics program under the OO and Functional programming spaces

Method set			
Method	**Argument(s) type**	**Return type**	**Use**
add	double, double	double	add(1,2) := 3
sub	double, double	double	sub(4,3) := 1
mul	double, double	double	mul(2,3) := 6
div	double, double	double	div(4,2) := 2
sqrt	double	double	sqrt(4) := 2
power	double, double	double	power(2,3) := 8
setValue	Settable, double	double	setValue(d,4) := d $\leftarrow$ 4
increment	Settable	double	d = 1, increment(d) := d $\leftarrow$ 2
addAndSet	Settable, double	double	d = 1, addAndSet(d,2) := d $\leftarrow$ 3
head	NList	double	$a = \{1, 2, 3\}$, head(a) := 1
tail	NList	NList	$a = \{1, 2, 3\}$, tail(a) := {2,3}
isEmpty	NList	boolean	$a = \{1, 2, 3\}$, isEmpty(a) := false
Conditional			
Control flow	**Argument(s) type**	**Return type**	-
IF-Then-Else	boolean, double, double	double	-
Terminal set			
Terminal	**Value**	**Type**	**-**
Constant	0.0, 1.0	double	-
Parameter[0]	-	double	-
Parameter[0]	-	NList	-

Experiment series $E_{Functional}$ use the ADF methodology, with static determination of program's architecture to automatically induce a program that exhibits the functionality specified in `FunctionalStatistics`. The only difference is that there is not a single *result producing branch* but instead different expression trees are being devoted for each dimension of the multi-task program. Furthermore, while Koza employs a simple module naming scheme to avoid the emergence of a circular hierarchy of calling dependencies, in this work, we impose no constraints on the hierarchical references between methods and allow each evolvable method to naturally call each other with no restrictions. It has been shown that GP has significant problems with scalability so a slightly more difficult problem becomes very much more difficult for GP. We are also interested in studying the scalability of simultaneous induction of the set of methods and we define two versions of

the original problem. These exhibit an incremental degree of difficulty by allowing only a subset of methods declared in the interfaces of figure 1 to be evolved in the evolutionary run of the first version. Version V_1 requires GP to evolve a program that computes the number of values in the statistical sample along with their mean and variance. Version V_2 builds on version V_1 and requires also the induction of the method that computes the standard deviation. Additionally, like in previous research, we treat the problem of automatic multi-task program induction as a multiobjective optimization problem and we employ multicriterion fitness functions. We perform a comparison between scalar and Pareto-based fitness assignment schemes. For the scalarization of multiple objectives, we employ no weighting schemes but allow for a plain sum of these. For Pareto-based fitness function we use the objective vector approach, by separating the performance of each evolvable method.

The primitive language contained elements that could be used from a human practitioner to implement a program that performs statistics. These are collectively presented in table 1. Standard arithmetic operations have been provided (`add`, `sub`, `mul`, `div`, `sqrt`, `power`), along with state manipulation operations (`setValue`, `increment`, `addAndSet`), list processing operations (`head`, `tail`, `isEmpty`), and an `If-Then-Else` statement that allows to control the flow of execution within a program.

The term *Evolutionary Programming* is preferred in this work since our EA uses a mutation-based variation operator to search the space of candidate solutions. *Subtree macromutation* (MM – substituting a node in the tree with an entirely randomly generated subtree of the same return type, under depth or size contraints) is the sole single-offspring variation operator applied to the population - no recombination is used. Experiments E_{OO_2} use an additional operator, *creation* (CR – a special case of mutation where an entirely new individual is created in the same way as in the initial random generation). The motivation for the *creation* operator lies on the fact that the number and type of instance variables defined in the initial sampling of *Evolvable Class* structures cannot be subsequently modified by the variation operator. Intuitively, CR guards against the premature loss of specifically configured packages of instance variables. Other than choosing the tree node to be replaced at random, we devise an additional, simple node selection scheme that allows us to select nodes at different depth levels using a uniform probability distribution, with the expectation to render bigger changes more likely.

Experiments that used a scalar fitness function employed a generational EA whether experiments with Pareto-based fitness function employed the NSGA-II algorithm [4]. For both algorithms, population size was set to 100 individuals and the number of generations was fixed to 1000. Their runs continued until an individual was generated that achieved a perfect score on the training data set or until all generations have elapsed. The maximum depth of a tree in the initial generation was set to 4 whereas the maximum depth resulting from the application of macromutation was set to 10. For the EA, a tournament size of 4 appeared to give efficient selection pressure and was combined with an elitism

scheme of 1%. NSGA-II used the *non dominated sorting* procedure combined with a binary tournament to perform selection of individuals [4]. In all experiments but E_{OO_2} macromutation was applied with 100% of probability. In E_{OO_2}, the creation operator was applied with a probability of 0.05%. Experiment series E_{OO} used the traditional approach of randomly choosing the tree-node to replace (choose a node from the whole tree uniformly) whether $E_{Functional}$ used a mixture of the traditional approach along with the additional node selection scheme previously presented. Their probability of application was set to 80% of selecting a node from within the whole tree and 20% of selecting a node from a particular depth.

The fitness evaluation of programs implementing the `OOStatistics` interface begins with the initialization of object state variables (all instance variables are set to zero). Then, the `addToSample` method is being invoked that many times to allow all values of the input data to be gradually passed as arguments to the method invocation. The changes made to the object state variables are maintained between subsequent `addToSample` invocations. Once all input data have been fed to the object the selector methods `n`, `mean`, `variance`, and `stdDeviation` are being sequentially invoked and a distance measure between actual and anticipated return values is computed. The distance measure takes the form of absolute error normalized over the [0, 1] interval with the value of zero representing the best possible fitness. On the other hand, the evaluation of a `FunctionalStatistics` program requires the sequential evaluation of each expression tree using the whole list of input data as a parameter. In both cases of program evaluation, training data consisted of 10 input lists of a maximum random length of 50. Test cases for generality consisted of 50 input lists of a maximum random length of 100. Elements were randomly chosen from the interval of [0, 1].

Furthermore, there is an additional significant issue that arises when evolving multi-tree programs, that of selecting which tree to choose to apply the variation operator. In this work we use a simple *brood selection* approach. Each time, macromutation is applied to produce 10 offspring from each evolvable method (i.e in the case of 5 evolvable methods macromutation would generate 50 offspring). The selection of points within a single tree is performed as discussed above. Parent trees that are believed to be correct (i.e have passed all training cases successfully) are being frozen from further modification.

Having specified our experimental methodology we went on to experiment with evolving an OO program as described in experiment series E_{OO_1}. Unfortunately OOEP was unable to induce an individual that correctly implements the statisticts program with best evolved individuals attaining an average training fitness of 0.11 and an average generalization fitness of 0.31. It was felt that this failure was part of the general difficulty of simultaneously inducing a set of methods, associated with an object, that cooperatively inspect and modify its internal state. Indeed, the successful induction of those methods that inspect the object state and base their computations on it cannot be performed in an enlightened way until the behavior of the state modification method is successfully

evolved. However, `addToSample` operates via its side-effects on the object state variables. Since we cannot measure its fitness directly, it can only be indirectly tested by observing if the other operations work correctly when called after it. It seems that, in this problem domain, this time-ordering of modifier and selector method invocations is not sufficient to successfully induce a modifier method that organizes the internal object memory in a useful way. To overcome this we needed to devise a strategy to reward the effect `addToSample` has on the object state variables. For this, we applied our knowledge of the problem and decided to add another two methods in `OOStatistics` and `FunctionalStatistics` respectively. These methods compute the sum and sum of squares of the values in the statistical sample.

For `FunctionalStatistics`, including the evolution of methods `sum(NList list)` and `sumSq(NList list)` is similarly beneficial as they can be used to express `mean(NList list)`, `variance(NList list)`, and `stdDeviation(NList list)`.

In addition, the space of constructible programs under the OO representation has been limited by placing restrictions upon which primitives could be used by which evolvable method. It is established good practice of the OO programming paradigm to classify the instance methods of a class into those that alter the state of the object and those that simply accesses it. In order to encourage the evolutionary process, the state manipulation primitives, presented in table 1, are only made available in the function set of the modifier method `addToSample()`. On the other hand, the space of selector methods allows for arithmetic computations based on the inspection of instance variables. The specific design parameters of each of the experiments are summarized in table 2.

Table 2. Experimental series

Experiment	Evolvable methods	State variables	Primitives used
$E_{OO_{1-V_1}}$	`addToSample`, `n`, `sum`, `sumSq`, `mean`, `variance`	3	arithmetic, state manipulation, constants, InstanceVariables, SettableVariables
$E_{OO_{1-V_2}}$	`addToSample`, `n`, `sum`, `sumSq`, `mean`, `variance`, `stdDev`	3	arithmetic, state manipulation, constants, InstanceVariables, SettableVariables
$E_{OO_{2-V_1}}$	`addToSample`, `n`, `sum`, `sumSq`, `mean`, `variance`	random, max 10	arithmetic, state manipulation, constants, InstanceVariables, SettableVariables
$E_{OO_{2-V_2}}$	`addToSample`, `n`, `sum`, `sumSq`, `mean`, `variance`, `stdDev`	random, max 10	arithmetic, state manipulation, constants, InstanceVariables, SettableVariables
$E_{Functional-V_1}$	`n`, `sum`, `sumSq`, `mean`, `variance`	n/a	arithmetic, list processing, constants, recursion allowed
$E_{Functional-V_2}$	`n`, `sum`, `sumSq`, `mean`, `variance`, `stdDev`	n/a	arithmetic, list processing, constants, recursion allowed

5 Results and Discussion

We performed 100 independent runs on each experiment of table 2 using both scalar and Pareto-based fitness functions. Table 3 presents the summary of the experimental results. Figure 3 presents `addSample` and `mean` methods from a sample evolved OO program. Notice how the settable variables are being updated by `addToSample` and their values being inspected by `mean`. First we present the probability of success (standard error in parentheses) of each different experimental setup. After Koza, *Min I(M,i,z)* represents a prediction of the minimum number of individuals that need to be evaluated in order to solve the problem with a probability of 99%. Average actual evaluations for successful and failed runs, as these were recorded during the evolutionary runs are also illustrated. These values include all evaluations resulting from brood formations. The average solution size is measured in terms of number of tree nodes in each successfully evolved individual. Looking at table 3 we observe that searching an OO programming space yields a higher probability of success. In cases of both scalar and Pareto-based fitness function this probability seems to be falling as we move down the table rows, from the experiments with the OO representation with preset object state, to those with random state and finally those with the functional representation. This probability is accompanied by the the predicted search size and the actual fitness evaluations required to yield a successful outcome. We note that these are being increased as we move from OO to functional program representation. Not surprisingly, the induction of recursive programs proved to be computationally more expensive.

Table 3. Summary of experimental results (standard errors in parentheses for prob. of success, avg. solution size in tree nodes)

Experiment	Prob. success (%)	Min. I(M,i,z)	Avg. actual evals. (success)	Avg. actual evals. (failure)	Avg. solution size
Scalar Fitness Function					
$E_{OO_{1-V_1}}$	4 (1.9)	471,200	76,275	2,121,860	44
$E_{OO_{1-V_2}}$	3 (1.7)	596,700	168,217	3,256,274	64
$E_{OO_{2-V_1}}$	2 (1.4)	5,517,600	1,636,535	2,312,276	36
$E_{OO_{2-V_2}}$	2 (1.4)	7,546,800	1,492,810	3,456,551	47
$E_{Functional-V_1}$	3 (1.7)	6,201,600	1,694,425	2,123,302	75
$E_{Functional-V_2}$	2 (1.4)	10,373,400	1,915,855	3,071,878	93
Pareto-based Fitness Function					
$E_{OO_{1-V_1}}$	4 (1.9)	775,200	146,230	2,318,375	47
$E_{OO_{1-V_2}}$	5 (2.2)	979,200	883,330	3,384,328	58
$E_{OO_{2-V_1}}$	1 (0.1)	1,285,200	1,955,323	2,612,850	64
$E_{OO_{2-V_2}}$	1 (0.1)	2,799,900	2,136,240	3,512,008	68
$E_{Functional-V_1}$	2 (1.4)	2,019,600	2,106,180	2,477,475	85
$E_{Functional-V_2}$	1 (0.1)	5,691,600	2,358,160	3,688,345	98

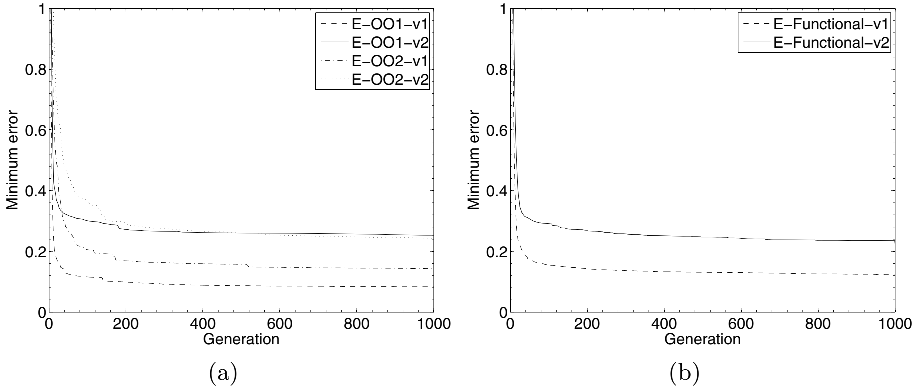


Fig. 2. Average of 100 runs of best-of-generation individuals represented in (a) OO, and (b) functional representation spaces, using a scalar fitness function

```
addToSample(Parameter[0])
(Method:mul
  (Method:increment
    SettableVariable[0])
  (Method:mul
    (Method:addAndSet
      SettableVariable[5]
      Parameter[0])
    (Method:addAndSet
      SettableVariable[1]
      (Method:mul Parameter[0] Parameter[0]))))
```

```
mean()
(Method:add
  (Method:div
    InstanceVariable[5]
    InstanceVariable[0]
  )
  Constant:0.0
)
```

Fig. 3. Sample evolved expression trees representing `addToSample` and `mean` respectively

As expected, the experiments with a fixed layout of object state variables were several orders of magnitude less computationally expensive than those required the organization of object state to be emergent throughout the evolutionary run. In addition, under both the OO and functional representations solving V_1 of the problem under consideration proved easier than solving V_2. Contrasting the performance of the evolutionary algorithm with scalar and Pareto-based fitness functions we initially observed that in terms of probability of success scalar fitness did slightly better than Pareto-based fitness for the OO experiments with a preordained layout of object state variables. The contrary appeared to be true for OO experiments involving self-organization of object memory and those using the functional representation. We note that on average the predicted search size using the Pareto-based fitness function is less than that computed for the scalar one. Surprisingly though, the actual computational effort required to induce a successful individual is greater. This means that while Pareto-based fitness function helps the EA to converge to the target solution in earlier generations, and so the predicted search size is smaller, it requires more fitness evaluations

stemming from the process of brood breeding. This excess number of fitness evaluations could be attributed to the general inefficiency of NSGA-II when dealing with more than two objectives.

Finally, we found that under an OO representation space, the EA was able to induce more parsimonious target individuals than the ones evolved under the functional representation.

6 Conclusions

Our hypothesis is empirically confirmed by comparing the success of the evolutionary search through the programming spaces defined by the object oriented and functional programming paradigms respectively. The experiments reported herein show that the simultaneous induction of a program's components can be more efficiently realized using the highly expressive representation offered by an OO program space. Setting the layout of object state in advance significantly increased performance, nevertheless we saw that the cooperative self-organization was also possible at a higher computational cost. For multiobjective fitness functions we found no significant difference in the probability of success rather than in the effort required to yield a successful run, rendering the use of scalar fitness more efficient. The identification of representation space is considered as one of the major human contributions to the GP learning mechanism. We strongly encourage future applications of GP to avoid the use of functional representation, wherever possible, and operate on an OO space.

References

1. Grandy Booch, *Object-Oriented Analysis and Design with applications*, Object Technology series. Addison Wesley, 2nd edition.
2. William B. Langdon, *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, vol. 1 of *Genetic Programming*, Kluwer, Boston, 24 April 1998.
3. Wilker Shane Bruce, "Automatic generation of object-oriented programs using genetic programming", in *Genetic Programming 1996: Proceedings of the First Annual Conference*.
4. Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA–II", *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, April 2002.

Evolving Modular Recursive Sorting Algorithms

Alexandros Agapitos and Simon M. Lucas

Department of Computer Science
University of Essex, Colchester CO4 3SQ, UK
aagapi@essex.ac.uk, sml@essex.ac.uk

Abstract. A fundamental issue in evolutionary learning is the definition of the solution representation language. We present the application of Object Oriented Genetic Programming to the task of coevolving general recursive sorting algorithms along with their primitive representation alphabet. We report the computational effort required to evolve target solutions and provide a comparison between crossover and mutation variation operators, and also undirected random search. We found that the induction of evolved method signatures (typed parameters and return type) can be realized through an evolutionary fitness-driven process. We also found that the evolutionary algorithm outperformed undirected random search, and that mutation performed better than crossover in this problem domain. The main result is that modular sorting algorithms can be evolved.

1 Introduction

A fundamental issue in evolutionary learning is the identification of the solution representation space. More specifically, given that the Genetic Programming (GP) paradigm relies on the evaluation of executable structures, the appropriate design of a primitive language is crucial. This language needs to embody a sufficient level of expressiveness for the desired phenotype to evolve. In traditional GP systems the representation system is composed of a static alphabet containing primitive terminal and non-terminal elements.

Traditional GP ignores much of what we know about how to design and implement well structured software, which to a significant practical degree, means object-oriented software. Indeed, much of the difficulty of high-level software design lies in the identification of useful abstractions. Building abstractions with procedures is arguably the main mechanism that conventional programming uses to address complex problems, and enables solutions to such problems to be specified as relatively simple compositions of sub-components. Past research has attempted to integrate modularity into the GP paradigm. Several approaches have been followed, including Automatically Defined Functions [1], Module Acquisition [2], Adaptive Representation through Learning [3], Automatically Defined Macros [4] and Structure Abstraction [5].

This paper presents work on coevolving general modular recursive sorting algorithms along with their representational language within an Object Oriented

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 301–310, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Genetic Programming System (OOGP). Sorting is a challenging problem for GP, and in general is not solvable with the usual GP-style constant time expression trees since the evolved algorithm will have to rearrange the comparable elements of sequences of arbitrary length into order. A literature review [6,7,8,9,10,11] on sorting algorithm evolution revealed a limited repertoire of attempts in this problem domain. While previous research on the evolution of iterative sorting has showed some promise, the evolution of recursive sorting algorithms has received very little attention from the evolutionary computation community, limited to the authors' previous work [7]. That study concentrated on evolving general recursive sorting algorithms. The time complexity of the successfully evolved algorithms was measured experimentally in terms of the number of method calls made, and for the best evolved individuals this was best approximated as $O(n \times log(n))$. Additionally, we investigated the effects of language design on evolving implementations of efficient sorting algorithms as well as the proficiency of five different fitness functions based on measures of sequence disorder.

2 Programming Space Under Exploration

On an intuitive level, the higher the complexity encapsulated in the primitive alphabet used to construct candidate solutions, the more expanded the class of problems that can be addressed. This prompts us to investigate a mechanism for adapting the primitive representational vocabulary by extending it with explicitly evolvable building blocks, tailored to the specific environment. This mechanism was first introduced in [1], under the name of *"evolutionary selection of program's architecture"* in an attempt to extend the *ADF* methodology by overcoming the need of pre-specifying the number of automatically defined functions (and their arguments), and the hierarchical references among them. Performance details of the application of this technique to a wide range of application areas are given in [1]. Here, we extend previous work of [1], under the notion of *"evolution of method signatures"* by simply adding *type* information in the return value and formal parameters of the evolvable member methods.

The evolutionary algorithm (EA) will be exploring the programming space of *sequences of comparable items*. For language primitives these are methods and objects of a simple, general-purpose list processing package presented in Table 1. Note that list-based classes and methods have been defined (using standard Java programming techniques). `CList` is a list of `Comparable` items, and `GreaterThan` and `NotGreaterThan` are predicates that implement the `MyComp` comparator interface. This interface declares a `Compare` method that compares `Comparable` items.

3 Evolvable Recursive Functions

It was shown in [7] that it is possible to reliably evolve a range of general recursive functions within an OOGP system. The recursion mechanism used is general and in-line with conventional programming's implementation of recursive calls.

Table 1. Primitive elements for evolving sorting algorithms

Method set		
Method	**Argument(s) type**	**Return type**
Head	CList	Comparable
Tail	CList	CList
Append	CList, CList	CList
Cons	Object, Object	CList
Compare	Comparable, Comparable	Boolean
EqualTo	Object, Object	Boolean
Conditional		
Control flow	**Argument(s) type**	**Return type**
IF-Then-Else	Boolean, CList, CList	CList
Terminal set		
Terminal	**Value**	**Type**
Parameter[0]	-	CList
Parameter[1]	-	MyComp
Parameter[2]	-	Comparable
Const: GreaterThan	new GreaterThan()	MyComp
Const: NotGreaterThan	new NotGreaterThan()	MyComp
Const: null	null	Object

It makes no distinction between built-in methods and the evolved method, thus making the evolved method's reference available to the method set serving as the alphabet for constructing the adaptive tree structures. Each evolved method in the OOGP system looks much like a Java method, with a declaration (signature: return type and parameter types) and an implementation, which is an expression tree evaluated with the arguments bound to the parameters. The expressions are strongly typed and may also invoke any specified methods in the Java API (as specified by the configuration of each experiment). In order to avoid the problem caused by non-terminating recursive structures we limited the recursive calls to between 25 and $10,500$. The upper bound of $10,500$ was chosen to be slightly larger than the largest number of recursive calls required by our hand-coded implementation of the most recursively expensive configuration, as discussed in [7]. In order to allow for the emergence of environment specific modules the hypothesis representation has been enhanced. The structure being evolved is now a *set of evolved methods*, reminiscent of a *Class* definition. For the sake of our discussion here we shall call this structure an *Evolvable Class*. The evaluation of such an individual begins from a pre-specified member method.

3.1 Evolutionary Selection of Hypothesis Structure

The syntactic structure of an *Evolvable Class* is dependent upon its constituent elements. However, in this work, the primitive set of elements is not static but includes a variable number of coevolving member methods. These methods in turn have variable signatures. When the initial random population is created,

it contains *Evolvable Classes* with different structures. That is, the number of evolvable member methods, and the number and type of arguments that they each possess differ from one individual to the other. The different *member method signatures* range over various useful instances. Each *Evolvable Class* is evaluated for fitness (starting from a pre-determined member method) and selected to participate in genetic operations using tournament selection.

3.2 Evolutionary Run Initialization

Each *Evolvable Class* has a pre-specified evolvable method that serves as the initial point of fitness evaluation. We call this *Main Member Method.* The signature of this member method is set a priori according to the signature of the target solution. The creation of an initial random *Evolvable Class* begins with the uniform random selection (from within a pre-specified range) of the number of the evolvable member methods (other than the *Main Member Method*) that will belong to it. Then a series of independent random choices is made for the number and type of arguments possessed by each member method. All of these random choices are made within a wide but limited range that includes every number and type that might be sensible for the problem at hand. We need to make clear that once the signatures of the evolvable member methods of an *Evolvable Class* are specified, they cannot be altered by applying a variation operator. The signature diversity enforced by the creation of the initial population plays a significant role in the success of the evolutionary run. Each evolvable member method (including the main one) allow recursive call to itself. Additionally, each member method is allowed to invoke hierarchically other methods of the *Evolvable Class.* A simple naming scheme has been employed to guard against circular calling dependencies.

3.3 Variation Operators

OOGP uses three main variation operators, namely, *macro-mutation* (MM — substituting a node in the tree with an entire randomly generated subtree with the same return type and a maximum random depth of 4 - subject to depth constraints), *creation* (CR — a special case of mutation where an entirely new individual is created in the same way as in the initial random generation) and *crossover* (XO). The motivation for the *creation* operator lies in the fact that method signatures are not modified after the creation of the initial population. CR guards against the premature loss of certain signatures.

The diversity of signatures of member methods among different *Evolvable Classes* has a concomitant impact to the mechanism of crossover. In order to guarantee that this variation operator will produce syntactically correct offspring, *Point Typing* has been used as in [1]. Our single-offspring crossover begins with the uniform selection of a member method from a contributing *Evolvable Class*. Subsequently, a point from the selected member method is uniformly chosen. The distribution of selection of crossover points is set to 90% probability of selecting interior nodes (uniformly) and 10% probability of selecting a leaf

node. The point from the receiving parental *Evolvable Class* is selected under the constraints of *Point Typing*. Crossover is then performed in the standard way. The resulting *Evolvable Class* inherits the member methods' signatures from the receiving *Evolvable Class*. During this process, as in [1], member methods coevolve with the *Main Member Method* resulting to the emergence of environment specific building blocks, advantageous to the composition of the final solution.

4 Experimental Context

Control parameters were specified as follows. Population size was set to $25,000$ individuals and the number of generations was fixed to 100. The maximum depth of a tree in the initial generation was set to 4 whereas the maximum depth resulting from the application of a variation operator was set to 10. We used three different search regimes to search the space of candidate solutions. The first regime, *XO-Regime*, used 95% XO, 4% MM and 1% CR. The second regime, *MM-Regime*, used 99% MM and 1% CR. Tournament selection (tournament sizes of 3 and 7 for *XO-Regime* and *MM-Regime* respectively) along with elitism (1%) was used as the selection scheme. Previous work [12] on the evolution of recursive and iterative algorithms has raised scepticism as to the degree that the performance of an evolutionary algorithm is not merely a result of a random exploration of the fitness landscape. It has been argued [13] that the space of algorithms is very discontinuous as to the space of functions, resulting in difficult to search landscapes, able to coerce the evolutionary learning process to be degenerated in a *needle-in-a-haystack* problem. In order to ensure that the ability to sort within our setup is not essentially a result of random search we are fixing an additional comparison between the EA and random search. This third regime, *RS-Regime*, used random search (RS) (i.e no selection pressure), but arranged in generations of purely random individuals (with a random maximum tree depth of 10) in order to plot the fitness on the same graphs as for the other search regimes.

The range of potentially useful numbers of member methods within a *Class* definition cannot be predicted with certainty for an arbitrary problem. The same holds for the range of their number of arguments. Here, we arbitrarily choose to use the number of 3 member methods, allowing a maximum of 4 evolvable member methods (including the main method) to be defined in an *Evolvable Class*. We set a sensible number of maximum arguments to an evolvable member method by inspecting the average number of arguments defined in methods from the *Java API* and also by inspecting a modular hand-coded recursive sorting algorithm implementation presented in [7]. Thus, we allow a maximum of three arguments to each method. For argument *types*, it is reasonable to draw possibly-useful instances from the programming space under consideration. Here we define the set S_{args} of possible argument types to be $S_{args} = \{\texttt{CList}, \texttt{Comparable}, \texttt{MyComp}\}$.

Five fitness functions based on different measures of sequence disorder were used as in [7]. These are: **(a)** *Mean Sorted Position Distance* **(MSPD)**, **(b)** *Mean Inversion Distance* **(MID)**, **(c)** *Minimum Number of Exchanges* **(MNE)**, **(d)**

Number of Step Downs **(NSD)**, **(e)** *Number of Elements to Remove* **(REM)**. The training cases consisted of 10 random lists of 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 unique elements respectively. Elements were randomly chosen from the range of $\{0, \ldots, 250\}$. Test sets measured the ability of an evolved solution to generalize to unseen data and recognized the success of a run. Test cases for generality consisted of 200 random lists (no element uniqueness requirement) with a maximum random length of 100. It is noteworthy that successfully evolved individuals were making no reference to the length of the input sequence and were subsequently tested correct with lists of up to 1000 elements in order to be evaluated for time complexity.

```
(sort(CList l)
  (EvolvableMethod
    (IF-Then-Else
      (EqualTo (l.Tail()) null)
      (l.Tail())
      (sort (l.Tail()))
    )
    Object: GreaterThan
    (l.Head())
  )
)
```

```
(EvolvableMethod(CList l, MyComp comp,
                 Comparator x)
  (If-Then-Else
   (comp.Compare(l.Head()) x)
   (Cons x l)
   (Cons (l.Head())
       (IF-Then-Else
        (EqualTo l null)
        (Cons x l)
        (EvolvableMethod (l.Tail()) comp x)
       )
)))
```

Fig. 1. Sample simplified evolved sorting algorithm

5 Evaluating the Generality of the Experimental Setup

On a practical level we want to ensure that the experimental setup is general and not biased toward sorting algorithms. For this purpose we used two supplementary experiments in order to evaluate the generality of the setup. The target functions were chosen to be those of (a) *reversing a list* (i.e. (reverse '(1 2 3) = (3 2 1)) and (b) *duplicating each element in a list* (i.e. (duplicate '(1 2 3) = '(1 1 2 2 3 3))). These recursive functions have the same signature as the sorting algorithm (accept a `CList` as an argument and return a `CList`). The idea is that by using the same primitive terminal and non-terminal sets but varying the fitness function and the training data we can lead the system to learn different target functions. Experiments used a population of 1000 individuals and 50 generations. MM (99%) and CR (1%) was the search regime employed. The fitness function was based on the sum of the positional distances between the same elements of the induced list and the target list, averaged over the length of the target list. Training and test set sizes, numbers of member methods, arguments, and argument types were set as above. We found that the probability of evolving target solutions for the *list reversal* problem was 94% (standard error: 2.4), resulting in a computational effort curve *I(M,i,z)* that reaches a minimum

value of 110,000 individuals in generation 5. Given that 10 fitness cases were used during training the number of fitness evaluations required is 1,100,000. Analogously, for the *list element duplication* problem, we got a probability of success of 81% (standard error: 3.9), resulting in an effort curve that reaches a minimum value of 400,000 (4,000,000 fitness evals.) by generation 9.

Table 2. Summary of results for each search regime on each fitness function (bold face indicates best performance on a given fitness function, standard errors in parentheses for prob. of success)

		Prob. of Success (%)	Minimum I(M,i,z)	Fitness Evaluations
MSPD	XO	6 (2.4)	138,750,000	1,387,500,000
	MM	**7 (2.5)**	**78,400,000**	**784,000,000**
	RS	0 (-)	-	-
MID	XO	1 (0.1)	378,675,000	3,786,750,000
	MM	**4 (1.9)**	**67,800,000**	**678,000,000**
	RS	0 (-)	-	-
MNE	XO	1 (0.1)	309,825,000	3,098,250,000
	MM	**4 (1.9)**	**98,800,000**	**988,000,000**
	RS	0 (-)	-	-
NSD	XO	1 (0.1)	321,300,000	3,213,000,000
	MM	**1 (0.1)**	**229,500,000**	**2,295,000,000**
	RS	0 (-)	-	-
REM	XO	1 (0.1)	413,100,000	4,131,000,000
	MM	**1 (0.1)**	**252,450,000**	**2,524,500,000**
	RS	0 (-)	-	-

6 Results and Discussion

We performed 100 independent runs, using each different search regime, in order to get statistically meaningful results. The computational effort *I(M,i,z)* was computed in the standard way, as described in [1]. Figures 2(a), 2(b), and 2(c) show the best-of-generation individuals of 100 independent runs using *XO-Regime*, *MM-Regime*, and *RS-Regime* respectively. Figures 2(d) and 2(e) provide a comparison of the cumulative probabilities of success between the different fitness functions under *XO-Regime* and *MM-Regime* respectively. Figure 2(f) presents a comparison of the average depth and size (in terms of number of nodes) of the successfully evolved individuals for the different fitness functions under *XO* and *MM* search regimes. Random search could not find any target solutions under any of the five fitness functions considered. Thus, in each run, 2,500,000 individuals (given 10 fitness cases results in 25,000,000 fitness evaluations) were processed without producing a general sorting algorithm.

Table 2, shows that the fitness function based on sequence disorder *MSPD*, performed consistently better under both variation operator regimes. This is

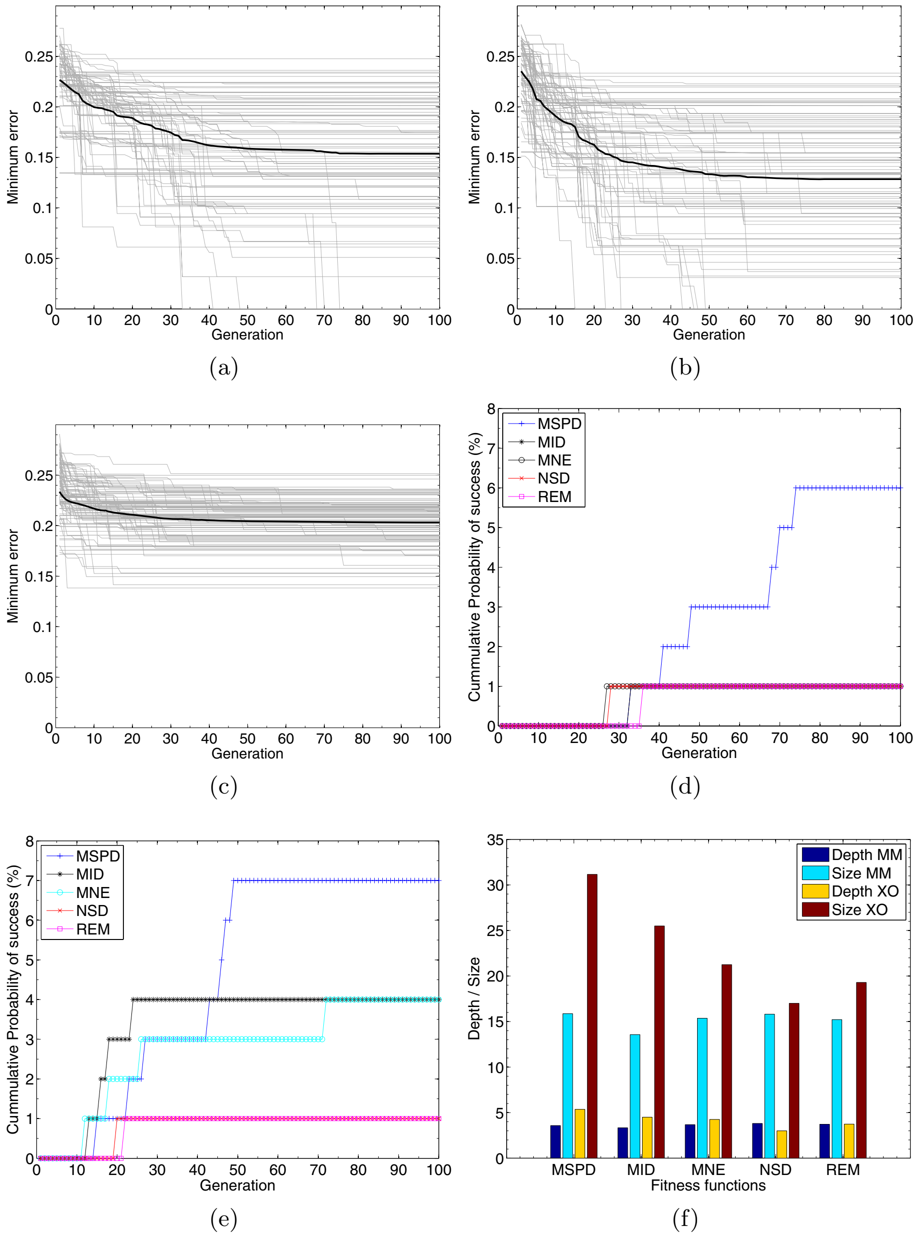


Fig. 2. (a) Best-of-generation individuals using *MSPD* and *XO-Regime*; (b) Best-of-generation individuals using *MSPD* and *MM-Regime*; (c) Best-of-generation individuals using *MSPD* and *RS-Regime*; Comparison of Cum. Prob. of success between different fitness functions using (d) *XO-Regime* and (e) *MM-Regime*; (f) Comparison of the average depth and size (in terms of number of nodes) of the successfully evolved sorting algorithms for the different fitness functions under *XO* and *MM* search regimes.

in-line with the previous results in [7] where *MSPD* and *MID* performed significantly better under that experimental setup. We also note that for *MSPD*, macro-mutation performed slightly better than crossover, however, the difference in their probability of success is rather insignificant. The important difference lies in the computational effort required to yield a successful outcome. Looking at the minimum error histograms in figures 2(a) and 2(b) we observe that the population under macro-mutation converges more rapidly, and this has a direct implication on the required fitness evaluations. The results presented in table 2 show that the superiority of macro-mutation in terms of parsimony in fitness evaluations is a general phenomenon as it remains essentially constant over all different fitness functions considered, and it becomes particulary significant in *MID* and *MNE*. Figures 2(d) and 2(e) present the performance curves under different variation operators. Looking at those graphs, we note that for *XO-Regime* most runs tend to stagnate after approximately generation 40 with the consistently better performance of *MSPD* stagnating after about generation 75. For *MM-Regime* we see a wider distribution of generation values for run stagnation, with *MNE* continuing evolution almost approximately up to generation 72. Observing the depth and size comparison we note that on average macro-mutation resulted in smaller solutions, mainly due to the additional depth constraint imposed during its application (the implanted subtree is not allow to grow past the depth of 4). Figure 1 presents a simplified sample evolved solution. We evaluate its efficiency in terms of method invocations required to sort sequences of up to 1000 elements. This is best approximated to $O(n^2)$, having a close fit to $F(n) = 1.255 \times n^2$. The coefficient (1.255) has been chosen that minimizes the mean squared error between $F(n)$ and estimated method invocations, for n being the length of the input sequence. We found that the algorithmic complexity has increased from $O(n \times log(n))$ in [7] to $O(n^2)$. Although we make no attempt to fully explain the results on a theoretical level, an intuitive understanding on the differing time efficiency of the evolvable recursive sorting algorithms under the three different experimental setups (as these are presented in [7] and the present paper) can be gained by considering two very important issues that were initially raised in [7]. First, this drop down in time efficiency could well hint at the inherent difficulty of inducing multi-tree structures. A second, most important issue, is related to the programming space explored by GP. This space is defined over all programs that can be constructed with the human-supplied primitive alphabet. The careful design of special language constructs, as done in [7], greatly enhanced the process of evolution and allowed GP to induce sorting algorithms of $O(n \times log(n))$ complexity. We empirically confirmed that in the absence of these special language constructs the time complexity of the successfully evolved algorithms was increased.

7 Conclusions

OOGP was successfully applied to the task of evolving modular recursive sorting algorithms. The evolved individuals were trained on small samples of data and

generalized perfectly. Evolution significantly outperformed random search. The feasibility of the process for automatically inducing the signatures of the representational building blocks was empirically justified. For that, a fitness function based on the positional distance between actual and sorted state performed the best. Beyond that, we believe that OOGP is an area with immense possibilities, including the evolution of complete classes, and cooperating sets of classes.

References

1. J.R. Koza, *Genetic Programming II: automatic discovery of reusable programs*, MIT Press, Cambridge, MA, (1994).
2. Peter J. Angeline and Jordan Pollack, "Evolutionary module acquisition", in *Proceedings of the Second Annual Conference on Evolutionary Programming*, 1993.
3. Justinian P. Rosca and Dana H. Ballard, "Discovery of subroutines in genetic programming", in *Advances in Genetic Programming 2*. MIT Press, 1996.
4. Lee Spector, "Simultaneous evolution of programs and their control structures", in *Advances in Genetic Programming 2*. MIT Press, 1996.
5. Tina Yu and Chris Clack, "Recursion, lambda abstractions and genetic programming", in *Genetic Programming 1998: Proceedings of the Third Annual Conference*.
6. Lee Spector, Jon Klein, and Maarten Keijzer, "The push3 execution stack and the evolution of control", in *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, New York, NY, USA, 2005, pp. 1689–1696.
7. Alexandros Agapitos and Simon M. Lucas, "Evolving efficient recursive sorting algorithms", in *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, Vancouver, 6-21 July 2006, pp. 9227–9234, IEEE Press.
8. Kenneth E. Kinnear, Jr., "Generality and difficulty in genetic programming: Evolving a sort", in *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, Stephanie Forrest, Ed., University of Illinois at Urbana-Champaign, 17-21 July 1993, pp. 287–294, Morgan Kaufmann.
9. Kenneth E. Kinnear, Jr., "Evolving a sort: Lessons in genetic programming", in *Proceedings of the 1993 International Conference on Neural Networks*, San Francisco, USA, 28 March-1 April 1993, vol. 2, pp. 881–888, IEEE Press.
10. Una-May O'Reilly and Franz Oppacher, "An experimental perspective on genetic programming", in *Parallel Problem Solving from Nature 2*, 1992.
11. Russ Abbott, Jiang Guo, and Behzad Parviz, "Guided genetic programming", in *The 2003 International Conference on Machine Learning; Models, Technologies and Applications (MLMTA'03)*, las Vegas, 23-26 June 2003, CSREA Press.
12. Scott Brave, "Evolving recursive programs for tree search", in *Advances in Genetic Programming 2*. MIT Press, 1996.
13. Astro Teller, "Genetic programming, indexed memory, the halting problem, and other curiosities", in *Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium*, Pensacola, Florida, USA, May 1994, pp. 270–274, IEEE Press.

Fitness Landscape Analysis and Image Filter Evolution Using Functional-Level CGP

Karel Slaný and Lukáš Sekanina

Faculty of Information Technology, Brno University of Technology
Božetěchova 2, 612 66 Brno, Czech Republic
slany@fit.vutbr.cz, sekanina@fit.vutbr.cz

Abstract. This work analyzes fitness landscapes for the image filter design problem approached using functional-level Cartesian Genetic Programming. Smoothness and ruggedness of fitness landscapes are investigated for five genetic operators. It is shown that the mutation operator and the single-point crossover operator generate the smoothest landscapes and thus they are useful for practical applications in this area. In contrast to the gate-level evolution, a destructive behavior of a simple crossover operator has not been confirmed.

1 Introduction

Cartesian Genetic Programming (CGP) was introduced by J. Miller and P. Thomson in 1999 [6]. In contrast with a standard genetic programming, CGP represents candidate programs as bounded $u \times v$-node directed graphs (rather than as trees), utilizes only a mutation operator and operates with a small population – to mention main differences.

In the connection with CGP, several issues have been discussed in the recent years, for example, the role of neutrality (which is implicit to the CGP representation) [2, 19], the role of bloat [9], modularity in CGP [17] and the usefulness of the search strategy which is based on a simple mutation [14, 8, 10]. While the standard GP benefits from crossover operators, it seems that a crossover is not useful for CGP at all (at least for the problems approached by CGP up to now). In other words, nobody has proposed a useful operator for CGP so far.

Techniques of fitness landscape analysis were utilized to obtain an information about CGP fitness landscapes in the digital circuit design task [14, 8, 16]. The structure of fitness landscapes has been studied in terms of their smoothness, ruggedness and neutrality on a time series obtained by sampling fitness values on a random walk. Easy problems are supposed to have smooth landscapes, while hard problems are supposed to be caused by rugged landscapes. It was recognized that the crossover is not a useful operator for evolving logic circuits using CGP, since the corresponding landscapes are extremely rugged. On the other hand, the mutation landscapes appear to be relatively smooth, and therefore, the mutation is more feasible for evolutionary circuit design [14].

In order to evolve more complicated digital circuits, CGP has been extended to operate at the functional level [12, 13]. Instead of simple gates, CGP works

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 311–320, 2007.
© Springer-Verlag Berlin Heidelberg 2007

with high-level components, such as adders, comparators, k-bit logic functions etc. This approach is especially well suited for the evolution of image operators, such as smoothing filters and edge detectors. It was reported in several case studies that the resulting operators are human-competitive [13, 5].

Similarly to the gate-level CGP, only a mutation operator was utilized for the functional-level CGP in mentioned applications. The problem investigated in this paper is whether a crossover operator is suitable for image operator evolution at the functional level using CGP. The motivation for investigating this problem comes from the following ideas: It seems that the image filter design problem is easier for CGP than, let us say, 3-bit multiplier design problem. Experimental results show that while approx. 30k generations are needed to find a good image filter [5], 5M generations are needed to find a good multiplier [15].[1]

The reason is that in case of gate-level circuit evolution, all possible input combinations are generated for a candidate circuit in order to obtain its fitness value. If only a subset of input combinations were considered, a vast majority of resulting circuits would not be fully functional and thus useful for real-world applications. On the other hand, the fitness function used in the image operator design problem utilizes only a subset of all possible combinations of image pixels. As human eyes are not able to see all the details in images, sufficiently good (not necessarily perfect) image operators are acceptable. It also seems that the mutation operator is more destructive for the multiplier evolution than for the filter evolution. This higher flexibility in resulting acceptable circuits and less destructive mutations indicate that corresponding fitness landscapes could be much smoother for the image filter evolution than for the gate-level evolution of multipliers.

The goal of this work is to perform the fitness landscape analysis for image filter design problem which is approached by the functional-level CGP. Various mutation and crossover operators will be compared in terms of fitness landscape analysis with the aim of identifying a suitable genetic operator for this particular problem. Regarding the previous analysis, our assumption is that a crossover operator can be identified which could help to make the resulting search algorithm more efficient than the original mutation-based approach.

2 Fitness Landscapes

The metaphor of fitness landscape, introduced in biology, expresses the idea that evolution can be viewed as a population flow on a surface in which the altitude of a point indicates how well the corresponding organism is adapted to an environment [18]. In the field of evolutionary computing, fitness landscapes are investigated in order to learn how difficult a particular problem is for a given search algorithm [4, 1, 8, 11]. On the basis of results of the fitness landscape

[1] Note that although the image filter evolution requires fewer generations, it takes much more time than the multiplier evolution, because the fitness value calculation is much more time consuming (e.g. 256 times more consuming for a training image of 128×128 pixels vs. a 3-bit multiplier with 64 test cases).

analysis, an original search algorithm is usually modified in order to improve its performance. Note that for GP, some authors have shown that the landscape metaphor may be deceptive [3].

A population moving on the fitness landscape creates a path which is called a walk on a fitness landscape. A walk $\{f_t\}_{t=0}^{n}$ can be described as a time series of $n+1$ fitness values which we acquired by using a reproduction operator.

Similarly to [14, 8] and in order to represent the changes made to the fitness value, we can introduce a string of marks which represent corresponding changes to the fitness value. Let $F = f_0 f_1 \dots f_n$ be a series of $n+1$ fitness values we have sampled during n generations from the initial population 0. This series can be represented by a string of symbols $S = s_1 s_2 \dots s_n$ over the alphabet $s_i \in \Sigma = \{\bar{1}, 0, 1\}$, which can be formalized by the function

$$\Psi_{f_t}(i,\varepsilon) = \begin{cases} \bar{1}, & f_i - f_{i-1} < -\varepsilon \\ 0, & |f_i - f_{i-1}| \leq \varepsilon \\ 1, & f_i - f_{i-1} > \varepsilon \end{cases} \tag{1}$$

so that

$$s_i = \Psi_{f_t}(i,\varepsilon) \tag{2}$$

for any constant ε [8]. For the fitness landscape $\mathcal{L}$ [8], the parameter ε is a real number from the interval $\langle 0, l_f \rangle$, where l_f is the greatest value on the fitness landscape. The parameter ε is used as an magnifying glass. The function $\Psi_{f_t}(i,0)$ is very sensitive to the changes made to the fitness value during the walk and thus the string $S(0)$ is set with the maximum accuracy. Contrariwise, the string $S(l_f)$ consists only of zeros.

By using the $S(\varepsilon)$ string, we can introduce two information characteristics called *entropic measures*. While the first entropic measure (FEM) is defined as

$$H(\varepsilon) = -\sum_{p \neq q} P_{[pq]} \log_6 P_{[pq]}, \tag{3}$$

the second entropic measure (SEM) is defined as

$$h(\varepsilon) = -\sum_{p = q} P_{[pq]} \log_3 P_{[pq]}. \tag{4}$$

The first measure rates the ruggedness of a landscape; the second measure estimates the smoothness of a landscape walk. The parameter $P_{[pq]}$ denotes the probability of the occurrence of the string pq in the string $S(\varepsilon)$. Smaller values of the measures mean neater walks.

Our goal is to evaluate different genetic operators by means of FEM and SEM. For the creation of the $S(\varepsilon)$ string we use a series of fitness values, which correspond to the fitness values of the best population member in each generation. As the parameter ε is set to 0, we can obtain the best resolution for the fitness landscape analysis.

3 CGP at the Functional Level for Image Filter Evolution

3.1 Cartesian Genetic Programming

In CGP, a candidate graph (circuit) is modeled as an array of u (columns) $\times$ v (rows) of programmable elements (gates). The number of circuit inputs, n_i, and outputs, n_o, is fixed. Feedback is not allowed. Each gate input can be connected to the output of a gate placed in the previous L columns or to some of circuit inputs. The L parameter, in fact, defines the level of connectivity and thus reduces/extends the search space. For example, if L=1 only neighboring columns may be connected; if $L = u$, the full connectivity is enabled. Each gate is programmed to perform one of functions defined in the set Γ. Figure 1 shows an example and a corresponding chromosome. Every individual is encoded using $u \times v \times 3 + n_o$ integers.

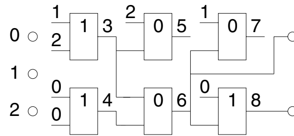


Fig. 1. An example of a 3-input circuit. CGP parameters are as follows: $L = 3$, $u = 3$, $v = 2$, $\Gamma = \{\text{AND (0), OR (1)}\}$. Gates 5 and 7 are not utilized. Chromosome: 1,2,1, 0,0,1, 2,3,0, 3,4,0 1,6,0, 0,6,1, 6, 8. The last two integers indicate the outputs of the circuit.

CGP operates with the population of λ individuals (typically, $\lambda = 5$). The initial population is randomly generated. Every new population consists of a parent (the fittest individual from the previous population) and its mutants. In case that two or more individuals have received the same fitness score in the previous population, the individual which did not serve as a parent in the previous population will be selected as a new parent. This strategy is used to ensure the diversity of population. The mutation operator modifies some randomly selected genes of an individual.

3.2 Image Filter Evolution

As introduced in [12, 13], every image operator will be considered as a digital circuit of nine 8-bit inputs and a single 8-bit output, which processes gray-scaled (8 bits/pixel) images. Fig. 2 shows that every pixel value of the filtered image is calculated using a corresponding pixel and its eight neighbors in the processed image. Each of circuit nodes can be programmed to perform one of functions given in Table 1. Recall that all functions operate with 8-bit operands and generate 8-bit outputs.

The goal of CGP is to propose a filter which minimizes the difference between filtered image I_2 and a reference image I_r which must exist for a particular

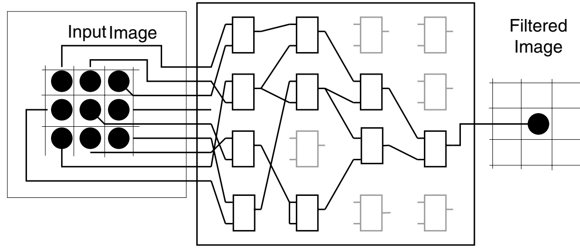


Fig. 2. Example of the image operator ($u = v = 4$). The output pixel value is calculated using the corresponding pixel and its eight neighbors in the input image.

Table 1. Functions used in circuit components. All functions have 8-bit operands and 8-bit outputs.

ID	Function	Description	ID	Function	Description
0	$x \vee y$	binary or	4	$x +_{sat} y$	addition with saturation
1	$x \wedge y$	binary and	5	$(x + y) >> 1$	average
2	$x \oplus y$	binary xor	6	$Max(x, y)$	maximum
3	$x + y$	addition	7	$Min(x, y)$	minimum

corrupted input image I_1. Suppose that the corrupted image and the reference image are of the size $K \times L$ pixels. Then the filtered image has the size of $(K-2) \times (L-2)$ pixels. The quality of the evolved image filter is evaluated by the fitness function

$$fitness = 255 \cdot (K-2) \cdot (L-2) - \sum_{i=1}^{K-2} \sum_{j=1}^{L-2} |I_2(i,j) - I_r(i,j)|. \qquad (5)$$

Papers [13, 5] show that this approach leads to very good image filters even in case that only a single image is utilized in the fitness function. As suitable image sizes, $K = 128$ and $L = 128$ are considered.

4 Proposed Genetic Operators

This section briefly introduces one mutation operator and four crossover operators we tested for image filter evolution. Please notice that operators that differ only in the number of offspring inserted into the new generation have quite significant differences in entropic measures.

Mutation Operator: CGP, as it is defined, uses only a mutation as the genetic operator. This operator selects the best-scored individual from the previous population. Its copies are mutated and inserted into the new population. Elitism is enabled. Diversity of the population is maintained according to the strategy described in Section 3.1.

```
do
  copy the best member into the new population
  mutate the just copied member in the new population
until the new population is not full
```

Single-Point Crossover (1): This is a standard one-point crossover operator which operates at the level of integer chromosomes. However, only one offspring of the crossover operation is mutated and included into the new population.

```
do
  choose two different best members of the previous population
  make one point crossover over its copies
  copy one offspring into the new population and mutate it
until the new population is not full
```

Single-Point Crossover (2): In comparison with the previous operator, this one copies the both offspring into the new population and mutates them.

Multi-point Crossover (1): This operator is an example of a multi-place crossover operator. This operator selects two (different) best members from the previous population. These two chromosomes are mutually combined in the way that they switch their functional blocks wiring; however, nodes' functions remain unchanged. Only one offspring is moved into the new generation. Then it is mutated.

```
do
  choose two different best members of the previous population
  make a multi-point crossover over its copies
  copy one offspring into the new population and mutate it
until new the population is not full
```

Multi-point Crossover (2): In comparison with the previous operator, this one copies the both offspring into the new population and mutates them.

5 Experimental Results

In order to estimate the smoothness and ruggedness of fitness landscapes, we measure the entropic measures h and H for each operator and parameters setting. Note that elitism is not utilized in our experiments.

The experiments are divided into two groups. While the first group operates with 7×4-node graphs (circuits) and 8-member population, the second group operates with 8×5-node graphs and population sizes 1, 2, 4 and 8 individuals. The mutation probability is set to 3% and L-parameter is set to 1 in both cases. A 128×128-pixel Lena image containing a random shot noise is utilized as a training image.

As these experiments are very time consuming, we use only 1000 generations in the first series of experiments (denoted as the *short run* in following figures).

In order to obtain more precise results, 100000 generations are performed in the second series of experiments (denoted as the *long run* in following figures). The first series is repeated 600 times; the second series is repeated 10 times.

Figure 3 shows average values of $H(0)$ (FEM) and $h(0)$ (SEM) for five genetic operators and for CGP with 7×4 nodes, i.e. for the first group of experiments. The lower average values the neater walk on the fitness landscape. Figures 4 and 5 show average values of $H(0)$ and $h(0)$ for the same genetic operators and CGP with 8×5 nodes. These figures are parameterized by the size of population.

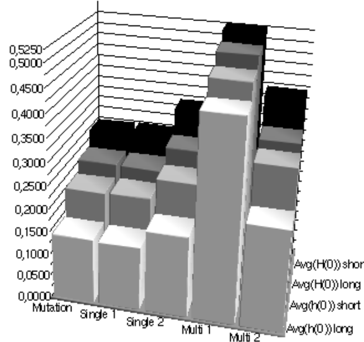


Fig. 3. Average values of $H(0)$ and $h(0)$ measures for five genetic operators and for CGP with 7×4 nodes

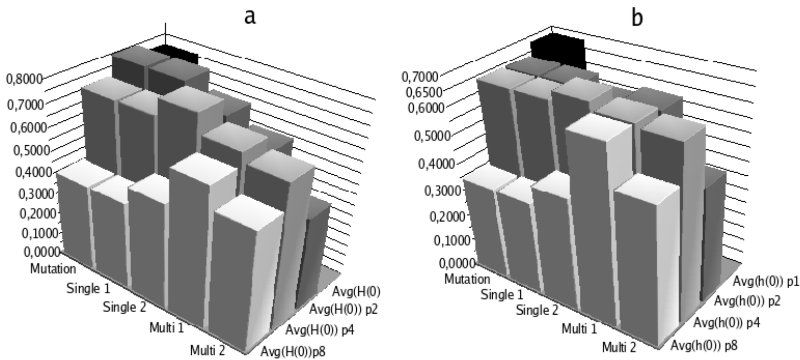


Fig. 4. Average values of $H(0)$ and $h(0)$ measures for five genetic operators and for CGP with 8×5 nodes. Results are given for *short runs* and population sizes 1, 2, 4 and 8 individuals (denoted as p1 – p8).

6 Discussion

For the first group of experiments, where the population contains 8 individuals, the mutation operator generates the smoothest fitness landscapes. Fig. 3

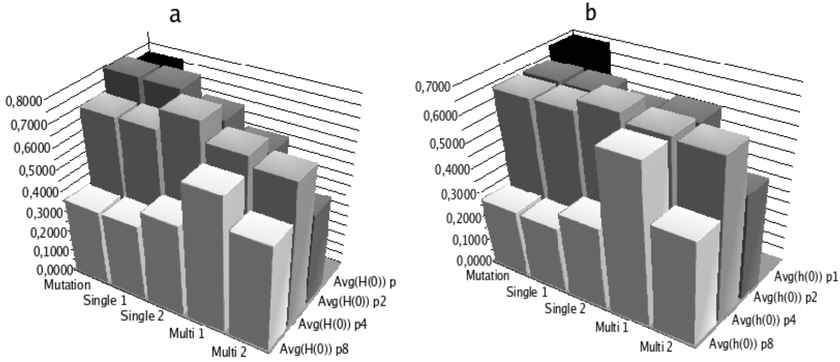


Fig. 5. Average values of $H(0)$ and $h(0)$ measures for five genetic operators and for CGP with 8×5 nodes. Results are given for *long runs* and population sizes 1, 2, 4 and 8 individuals (denoted as p1 – p8).

Fig. 6. Left – Lena image corrupted by the shot noise. Middle – resulting image for the multi-point crossover operator (1). Right – resulting image for the mutation operators.

also shows that Single-point Crossover (1) produces practically identical results. Other crossover operators generate more rugged fitness landscapes.

For the second group of experiments, we can observe that the shape of bars remains practically unchanged (for the population size of 8 individuals) when compared with the shape obtained for the first group. However, the average values of $H(0)$ and $h(0)$ are slightly higher, which is caused by using more nodes in CGP. By increasing the number of generations, the average values of the entropic measures slightly decrease; however, only when the population consists of 8 individuals. The number of generations does not influence the average values of $H(0)$ and $h(0)$ for remaining genetic operators significantly.

We can observe that the population size is the parameter which significantly contributes to the average values of the entropic measures. The mutation operator produces the smoothest landscapes for the population of 8 individuals. For smaller populations, crossover operators are becoming more valuable, probably because they are able to introduce more diversity to the search process. Recall that standard version of CGP usually operates with the population of 5 individuals. Our results confirm that the use of four or less individuals is not advantageous. We can also observe different behaviors of crossover operators we

have investigated. For the search process it is beneficial when the single point crossover provides only one mutated offspring to the new population; on the other hand, the both offspring should be used in case of the multi-point crossover.

Examples of resulting Lena images (see Fig. 6) were obtained by the mutation operator and the multipoint crossover (1) operator, with an 8-member population and after 100000 generations. Note that the best filters reported in [13] are able to remove this type of noise perfectly.

Approx. 22 hours of computation of a 1.3 GHz CPU are needed to finish a single run of CGP (100000 populations were produced on the topology of 8×5-node graphs, for 128×128-pixel images and with 8-member population).

7 Conclusions

A fitness landscape analysis was performed for the image filter design problem approached using functional-level CGP. Smoothness and ruggedness of fitness landscapes were measured for five genetic operators. It was shown that the mutation operator and the single-point crossover operator generate the smoothest landscapes and thus they are useful for practical applications in this area. In contrast to the gate-level evolution, a destructive behavior of a simple crossover operator was not confirmed. As the mutation operator is easy to implement, it remains the most useful operator for image filter evolution using CGP.

Acknowledgements

This work was supported by the Grant Agency of the Czech Republic under No. 102/07/0850 *Design and hardware implementation of a patent-invention machine* and the Research intention No. MSM 0021630528 – Security-Oriented Research in Information Technology.

References

[1] Alander J. T.: Population size, building block, fitness landscape and genetic algorithm search efficiency in combinational optimisation: An empirical study. Practical Handbook of Genetic Algorithms, Vol. 3, CRC Press, 1999, p. 459–485
[2] Collins, M.: Finding needles in haystacks is harder with neutrality. Genetic Programming and Evolvable Machines. Vol. 7, No. 2, 2006, p. 131–144
[3] Daida J. M. et al.. What makes a problem GP-hard? Genetic Programming and Evolvable Machines. Vol. 2, No. 2, 2001, p. 165–191
[4] Jones, T.: Evolutionary Algorithms, Fitness Landscapes and Search. PhD thesis, University of New Mexico, 1995, p. 249
[5] Martinek, T., Sekanina, L.: An evolvable image filter: Experimental evaluation of a complete hardware implementation in FPGA. In: Proc. of the 6th International Conference Evolvable Systems: From Biology to Hardware, ICES 2005, Sitges, Barcelona, LNCS 3637, Springer-Verlag, 2005, p. 76–85

[6] Miller, J., Thomson, P.: Cartesian genetic programming. In: Proc. of the 3rd European Conference on Genetic Programming, LNCS 1802, Springer Verlag, Berlin, 2000, p. 121–132
[7] Miller, J., Job, D., Vassilev, V.: Principles in the evolutionary design of digital circuits – Part I. Genetic Programming and Evolvable Machines. Vol. 1., No. 1, 2000, p. 8–35
[8] Miller, J., Job, D., Vassilev, V.: Principles in the evolutionary design of digital circuits – Part II. Genetic Programming and Evolvable Machines. Vol. 1, No. 2, 2000, p. 259–288
[9] Miller, J.: What bloat? Cartesian Genetic Programming on Boolean Problems. In: GECCO'01 - Late Breaking Papers. Proc. of the 3rd Genetic and Evolutionary Computation Conference, San Francisco, CA, Morgan Kaufmann Publishers, 2001, p. 295–302
[10] Miller, J., Smith, S.: Redundancy and Computational Efficiency in Cartesian Genetic Programming. IEEE Transactions on Evolutionary Computation. Vol. 10, No. 2, 2006, p. 167–174
[11] Reeves, C.: Fitness Landscapes. In: Burke, E.K. and Kendall, G., Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques, Springer 2005, pp. 587-610.
[12] Sekanina, L.: Image Filter Design with Evolvable Hardware. In: Applications of Evolutionary Computing – Proc. of the 4th Workshop on Evolutionary Computation in Image Analysis and Signal Processing EvoIASP'02, LNCS 2279, Springer-Verlag, Berlin, 2002, p. 255–266
[13] Sekanina, L.: Evolvable components: From Theory to Hardware Implementations, Springer-Verlag, Natural Computing Series, 2004
[14] Vassilev, V., Miller, J., Fogarty, T.: On the Nature of Two-Bit Multiplier Landscapes. In: Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware, Pasadena, CA, IEEE Computer Society, Los Alamitos, 1999, p. 36–45
[15] Vassilev, V., Job, D., Miller, J.: Towards the Automatic Design of More Efficient Digital Circuits. In: Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware, CA, USA IEEE Computer Society, Los Alamitos, 2000, p. 151–160
[16] Vassilev, V., Miller, J.: Scalability Problems of Digital Circuit Evolution. In: Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware, CA, USA, IEEE Computer Society, Los Alamitos, 2000, p. 55–64
[17] Walker, J., Miller, J.: Investigating the performance of module acquisition in cartesian genetic programming. Proc. of GECCO 2005, ACM, 2005, p. 1649–1656
[18] Wright, S.: The roles of mutation, inbreeding, crossbreeding, and selection in evolution. In: Proceedings of the 6th Int. Congress on Genetics, 1932, p. 355–366
[19] Yu, T., Miller, J.: Neutrality and the Evolvability of Boolean Function Landscape. Proc. of the 4th European Conference on Genetic Programming, LNCS 2038, Springer Verlag, Berlin, 2001, p. 204–217

Genetic Programming Heuristics for Multiple Machine Scheduling

Domagoj Jakobović, Leonardo Jelenković, and Leo Budin

University of Zagreb, Faculty of Electrical Engineering and Computing, Croatia
{domagoj.jakobovic,leonardo.jelenkovic,leo.budin}@fer.hr

Abstract. In this paper we present a method for creating scheduling heuristics for parallel proportional machine scheduling environment and arbitrary performance criteria. Genetic programming is used to synthesize the priority function which, coupled with an appropriate meta-algorithm for a given environment, forms the priority scheduling heuristic. We show that the procedures derived in this way can perform similarly or better than existing algorithms. Additionally, this approach may be particularly useful for those combinations of scheduling environment and criteria for which there are no adequate scheduling algorithms.

1 Introduction

Scheduling is concerned with the allocation of scarce resources to activities with the objective of optimizing one or more performance measures, which can assume minimization of makespan, job tardiness, number of late jobs etc. Due to inherent problem complexity and variability (most of the real-world scheduling problems are NP complete), a large number of scheduling systems employ heuristic scheduling methods. Given different performance criteria and user requirements, the question arises as to which heuristic to use in a particular environment? The problem of selecting an appropriate scheduling policy is an active area of research [1][2] and a considerable effort is needed to choose or develop the algorithm best suited to the given environment. An answer to this problem may be provided using machine learning methods to create problem specific scheduling algorithms.

The combinatorial nature of most scheduling problems allows the use of search based and enumerative techniques [1], such as genetic algorithms, branch and bound, simulated annealing, tabu search etc. These methods usually offer good quality solutions, but at the cost of a large amount of computational time. Search based techniques are hence not applicable in dynamic or uncertain conditions where there is a need for frequent schedule modification or reaction to changing system requirements (i.e. resource failures or job parameter changes). Scheduling with fast heuristic algorithms is therefore highly effective, and the only feasible solution, in many instances.

In this paper we describe a methodology for evolving scheduling heuristics with genetic programming (GP). Genetic programming has rarely been employed

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 321–330, 2007.
© Springer-Verlag Berlin Heidelberg 2007

in scheduling, mainly because it is impractical to use to search the space of potential solutions (i.e. schedules). It is, however, very suitable for searching the space of *algorithms* that provide solution to the problem. Previous work in this area of research includes evolving scheduling policies for the single machine unweighted tardiness problem [3][4][5], single machine scheduling subject to breakdowns [6], classic job shop tardiness scheduling [7][8] and airplane scheduling in air traffic control [9][10]. The scheduling procedure in those papers is however defined only implicitly for a given scheduling environment. In this paper we structure the scheduling algorithm in two components: a meta-algorithm which uses priority values to perform scheduling and a priority function which defines values for different elements of the system. To illustrate this technique we develop scheduling heuristics for multiple proportional machine environment (described in the next section) for which a methodology with GP has, to the best of our knowledge, not been published previously. We also include several combinations of additional requirements, such as dynamic job arrivals, sequence dependent setup times and different scheduling criteria.

2 Parallel Machine Environment

2.1 Problem Statement

In a parallel machine environment, a number n of jobs J_j compete for processing on either of m machines. In a static problem each job is available at time zero, whereas in a dynamic problem each job has a release date r_j. The nominal processing time of the job is p_j and its due date is d_j. Each machine in the system has a speed s_i so that the actual processing time of job j on a machine i is given with $p_{ij} = p_j/s_i$. Relative importance of a job is denoted with its weight w_j. The most widely used scheduling criteria for this environment include weighted tardiness, number of tardy jobs, flowtime and makespan. If C_j denotes the finishing time of job j, then the job tardiness T_j is defined as

$$T_j = \max\{C_j - d_j, 0\} \quad . \tag{1}$$

Lateness of a job U_j is taken to be 1 if a job is late, i.e. if its tardiness is greater than zero, and 0 otherwise. Flowtime of a job is the time the job has spent in the system, i.e. the difference between job release time and completion time: $F_j = C_j - r_j$, whereas the makespan (C_{max}) is the maximum finishing time of all the jobs in a set. Based on these output values, the weighted scheduling criteria are defined as follows: weighted tardiness for a set of jobs is defined as

$$T_w = \sum\nolimits_j w_j T_j \quad , \tag{2}$$

weighted number of late jobs as

$$U_w = \sum\nolimits_j w_j U_j \quad , \tag{3}$$

and weighted flowtime as

$$F_w = \sum\nolimits_j w_j F_j \quad . \tag{4}$$

In the case where a machine may need to process more than one type of job, there is sometimes the need to adjust the machine for the processing of the next job. If the time needed for adjusting depends on the previous and the following job, this is referred to as *sequence dependent setup time* and must be defined for every possible combination of two jobs [11] [12]. This condition further increases the problem complexity for some scheduling criteria.

In the evolution process, a single scheduling criteria can be selected as fitness function where smaller values indicate greater fitness. The total quality estimate of an algorithm is expressed as the sum of criteria values over all the test cases.

2.2 Test Cases Formulation

Each scheduling instance is defined with the following parameters: the number of machines m and their speeds s_i, the number of jobs n, their nominal processing times p_j, due dates, release dates and weights. The values of processing times are generated using uniform, normal and quasi-bimodal probability distributions among the different test cases. The number of jobs varies from 12 to 100 and number of machines from 3 to 20. With machine speeds we can define the effective number of machines $\hat{m}$ as the sum of speeds of all machines:

$$\hat{m} = \sum_{i=1}^{m} s_i \quad , \tag{5}$$

where m represents the actual number of machines.

In some of the test environments we allow for the job sequence dependent setup times. A distinct setup time, which does not depend of the speed of the machine, is defined for every possible sequence of two jobs. The values of all of the above parameters are generated in accordance with methods and examples given in [4], [11], [12] and [13]. Overall, we defined 120 test cases for learning and 600 evaluation test cases for comparison of the evolved and existing scheduling heuristics.

2.3 Scheduling Heuristics

The scheduling method investigated in this work is priority scheduling, in which certain elements of the scheduling system are assigned priority values. The choice of the next activity being run on a certain machine is based on their respective priority values. This kind of scheduling algorithm is also called, variously, 'dispatching rule', 'scheduling rule' or just 'heuristic'. The term scheduling rule, in a narrow sense, often represents only the *priority function* which assigns values to elements of the system (jobs in most cases). For instance, a scheduling process may be described with the statement 'scheduling is performed using EDD rule'.

While in most cases the method of assignment of jobs on machines based on priority values is self-evident, in some environments it is not. This is particularly true in dynamic conditions where jobs arrive over time or may not be run before some other job finishes. That is why a *meta-algorithm* must be defined for each scheduling environment, dictating the way activities are scheduled based on their priorities and possible system constraints. The meta-algorithm encapsulates the priority function, but the same meta-algorithm may be used with different priority functions and vice versa [14]. In virtually all the literature on the subject the meta-algorithm part is never explicitly expressed but only presumed implicitly, which can lead to many misunderstandings between different projects.

The time complexity of priority scheduling algorithms depends on the meta-algorithm, but it is in most cases negligible compared to search-based techniques, which allows the use of this method in on-line scheduling [15] and dynamic conditions. All the heuristics presented in this paper, including the evolved ones, provide a solution for several hundred instances in less than a second (since the priority functions are evolved offline).

In this work, we included the following widely used scheduling heuristics for efficiency comparison: weighted shortest processing time (WSPT), earliest due date (EDD), longest processing time (LPT), X-dispatch bottleneck dynamics heuristic [13] (XD), Rachamadugu & Morton heuristic [16] (RM), weighted Montagne heuristic [13] (MON) and Apparent Tardiness Cost with Setups heuristic [11] (ATCS). Each heuristic is defined with its priority function which is used by a meta-algorithm for a given environment (stated in the next section).

3 Scheduling with Genetic Programming

In this work we use the described elements of priority scheduling paradigm, so that the meta-algorithm part is defined manually for a specific scheduling environment and the priority function is evolved with genetic programming using appropriate functional and data structures. This way, using the same meta-algorithm, different scheduling algorithms best suited for the current criteria can be devised. The task of genetic programming is to find such a priority function which would yield the best results considering given meta-algorithm and user requirements. The solution of genetic programming is represented with a single tree that embodies the priority function. After the learning process, single best found priority function is tested on all evaluation test cases and compared with existing heuristics. Following the described priority scheduling procedure, we define the following meta-algorithm which is used with all the existing heuristics as well as with GP evolved priority function for static job availability:

while there are unscheduled jobs **do**
  wait until a machine (k) is ready;
  calculate priorities of all available jobs on machine k;
  schedule job with best (greatest) priority on machine k;
end while

Handling Dynamic Job Arrivals. In a dynamic environment the scheduler can use algorithms designed for a static environment, but two things need to be defined for those heuristics. The first is the subset of the jobs to be taken into consideration for scheduling, since some jobs may arrive in some future moment in time. The second issue is the method of evaluation of jobs which have not yet arrived, i.e. the question should the priority function for those jobs be different and in what way. This can be resolved in the following ways:

1. no inserted idleness - we only consider jobs which are immediately available;
2. inserted idleness - waiting for a job is allowed and waiting time is added to job's processing time in priority calculation;
3. inserted idleness with arbitrary priority - waiting is allowed but the priority function must be defined so that it takes waiting time into account.

When using existing heuristics for comparison, we apply the second approach where necessary, i.e. if the priority function does not take job's release date into account. Genetic programming, on the other hand, is coupled with the third approach, as it has the ability to learn and make use of waiting time information on itself. Scheduling heuristics that presume all the jobs are available are modified so that the processing time of a job includes job's time till arrival (waiting time), denoted with

$$wt_j = \max \{r_j - time, 0\} \ . \tag{6}$$

Thus, if an algorithm uses only the processing time of a job, that time is increased by wt_j of the job. All the described heuristics, except the XD heuristic which is defined for a dynamic environment, are modified in this manner when solving the dynamic variant of the scheduling problem.

The question remains as to which jobs to include when calculating the priority function? It can be shown that, for any regular scheduling criteria [13], a job should not be scheduled on a machine k if the waiting time for that job is longer than the processing time of the shortest of all currently available unscheduled jobs on that machine (some scheduling software implementations also include this condition [17]). In other words, we may only consider jobs j for which

$$wt_j < \min_i \{p_{ki}\}, \forall i : r_i \leq time \ . \tag{7}$$

This approach can be illustrated with the following meta-algorithm which is used in dynamic conditions with an arbitrary priority function:

while there are unscheduled jobs **do**
  wait until a machine (k) and at least one job are ready;
  p_{MIN} = processing time of the shortest available job on machine k;
  calculate priorities of all jobs j with $wt_j < p_{MIN}$;
  schedule job with best priority;
end while

Table 1. The genetic programming parameters

Parameter / operator	Value / description
population size	10000
max. individual depth	17
selection	steady-state, tournament of size 3
stopping criteria	maximum number of generations (150) or maximum number of consecutive generations without best solution improvement (30)
crossover	85% probability, standard crossover
mutation	standard, swap and shrink mutation, 3% probability each
reproduction	5% probability
initialization	ramped half-and-half, max. depth of 5

Handling Sequence Dependent Setups. Almost any heuristic may be adjusted to include sequence dependant setup time with a method presented in [13]. The job priority obtained with the original function is decreased by a certain value that measures the additional cost brought by setup time for that job. If the original priority value is denoted with π_j, then the priority with setup times is given with

$$\pi_{lj} = \pi_j - \frac{s_{lj}}{(p_{AV}/\hat{m}) \cdot (p_j/s_k)} \ , \tag{8}$$

where l is the last processed job, s_{lj} setup time between job l and job j and p_{AV} the average nominal processing time of all unscheduled jobs. All existing heuristics are modified in this way when solving for setup times, except the ATCS heuristic which is specifically designed for this scheduling condition.

Genetic Programming Parameters, Functions and Terminals. The GP parameters used are presented in Table 1. We did not experiment with many parameter combinations as the GP efficiency did not vary noticably in respect to scheduling heuristic efficiency. The most crucial decision is finding the minimal set of functions and terminals that will satisfy the sufficiency property for a given environment. We define the same function set for every scheduling environment and a different terminal set depending on the variant of the problem (for sequence dependent setups and/or dynamic job arrivals). The complete set, along with guidelines for terminal usage, is given in Table 2.

3.1 Scheduling with Static Job Availability

In a static environment all jobs (and all machines) are available at time zero. The task of genetic programming is to evolve such a priority function that would produce schedules of a good quality for a given performance criteria. We made two sets of experiments: one for the simple static problem and another with additional sequence dependent setups, both optimizing minimum weighted tardiness criteria.

Table 2. The function and terminal set

Function name	**Definition**
ADD, SUB, MUL	binary addition, subtraction and multiplication operators
DIV	protected division: DIV $(a, b) = \begin{cases} 1, \text{if } \lvert b \rvert < 0.000001 \\ a/b, \text{otherwise} \end{cases}$
POS	POS $(a) = \max\{a, 0\}$
Terminal name	**Definition**
Terminals used in every problem variant	
pt	nominal processing time of a job (p_j)
dd	due date (d_j)
w	weight (w_j)
Nr	number of remaining (unscheduled) jobs
SPr	sum of processing times of remaining jobs
SD	sum of due dates of all jobs
SL	positive slack, $\max\{d_j - p_j - time, 0\}$
SLs	positive slack using machine speed, $\max\{d_j - p_j/s_k - time, 0\}$
SPD	speed of the current machine (s_k)
Msm	the sum of all machine speeds (effective number of machines, $\hat{m}$)
Terminals for sequence dependent setups	
STP	setup time from previous to job j
Sav	average setup time from previous (l) to all jobs $\frac{1}{n-1}\sum_{j=1}^{n} s_{lj}$
Terminals for dynamic environment	
AR	job arrival time (waiting time), $\max\{r_j - time, 0\}$

For the first set (notation $Q\,||\sum w_j T_j$ in scheduling theory) we conducted 20 runs and achieved mean best result of 37.6 with std. deviation $\sigma = 1.38$ in weighted tardiness as fitness function on evaluation set of 600 unseen test cases. Apart from total criteria values, a good performance measure for a scheduling heuristic may be defined as the percentage of test cases in which the heuristic provided the best achieved result (or the result that is not worse than any other heuristic). This value can be denoted as the dominance percentage. Both types of results are shown in the uppermost section of Table 3 and best results in each category are marked in boldface.

It can be noted that the performance is mainly divided between different heuristics: GP evolved heuristic achieved best weighted tardiness result, WSPT rule best weighted flowtime and LPT best makespan. Another set of 20 runs was conducted in the same environment but with the inclusion of sequence dependent setups (notation: $Q\,|s_{ij}|\sum w_j T_j$) and additional GP terminals from Table 2. The best solutions were found with the mean 42.7 and $\sigma = 2.5$. The results for this variant are shown in Table 3. It is clear from the results that in this environment the GP evolved heuristic obtained very good performance over more criteria.

Table 3. Scheduling criteria values and dominance percentages

Heuristic	**Scheduling criteria**				**Dominance percentage**			
	Twt	**Uwt**	**Fwt**	**Cmax**	**Twt**	**Uwt**	**Fwt**	**Cmax**
Static job arrivals, weighted tardiness optimization								
GP	**34.1**	28.1	41.1	90.3	**79 %**	21 %	7 %	11 %
RM	46.2	27.4	41.7	92.1	14 %	22 %	1 %	6 %
MON	46.2	**25.0**	36.0	92.9	8 %	27 %	24 %	5 %
WSPT	49.8	25.1	**35.1**	94.4	1 %	**31 %**	**66 %**	3 %
EDD	66.1	36.0	41.3	92.9	4 %	9 %	2 %	6 %
LPT	115.7	44.7	52.6	**83.6**	0 %	0 %	0 %	**73 %**
Static job arrivals, sequence dependent setups, weighted tardiness optimization								
GP	**42.1**	**38.4**	**63.2**	69.4	**87 %**	**75 %**	**72 %**	20 %
ATCS	61.0	44.7	68.7	70.6	9 %	22 %	22 %	9 %
RM	76.4	50.1	78.2	69.7	1 %	12 %	1 %	10 %
WSPT	71.6	47.6	72.9	**66.8**	1 %	15 %	4 %	**45 %**
MON	73.8	49.0	76.7	68.7	2 %	13 %	1 %	15 %
LPT	85.8	52.7	83.2	74.4	1 %	16 %	1 %	2 %
Dynamic job arrivals, weighted tardiness optimization								
GP	**33.0**	**23.9**	26.7	47.9	**78 %**	**45 %**	9 %	11 %
XD	39.3	26.5	28.0	48.5	10 %	18 %	5 %	10 %
MON	39.3	24.6	25.0	48.7	5 %	19 %	29 %	11 %
WSPT	41.1	24.3	**24.5**	48.7	3 %	27 %	**54 %**	9 %
EDD	50.2	33.0	27.6	47.8	6 %	6 %	4 %	13 %
LPT	81.8	39.4	34.9	**44.9**	2 %	4 %	2 %	**70 %**
Dynamic job arrivals, makespan optimization								
GP	78.7	39.1	34.2	**41.9**	5 %	2 %	7 %	**68 %**
XD	**39.3**	26.5	28.0	48.5	**39 %**	24 %	6 %	6 %
MON	**39.3**	24.6	25.0	48.7	33 %	35 %	30 %	8 %
WSPT	41.1	**24.3**	**24.5**	48.7	16 %	**45 %**	**55 %**	6 %
EDD	50.2	33.0	27.6	47.8	9 %	9 %	3 %	8 %
LPT	81.8	39.4	34.9	44.9	2 %	3 %	1 %	40 %
Dynamic job arrivals, sequence dependent setups, weighted tardiness optimization								
GP	**51.1**	**47.5**	**71.7**	87.4	**92%**	**72%**	**86%**	19%
ATCS	67.8	49.8	81.2	91.5	4%	45%	9%	8%
XD	78.1	53.6	87.8	85.7	2%	27%	2%	17%
WSPT	75.7	51.7	84.6	**84.0**	1%	30%	2%	31%
MON	78.3	52.6	87.4	85.7	2%	28%	2%	20%
LPT	81.9	54.2	88.8	85.0	1%	27%	1%	**35%**
Dynamic job arrivals, sequence dependent setups, makespan optimization								
GP	**53.9**	50.2	**65.1**	**73.9**	**89%**	56%	**95%**	**89%**
ATCS	67.8	**49.8**	81.2	91.5	8%	**62%**	3%	7%
XD	78.1	53.6	87.8	85.7	1%	27%	0%	9%
WSPT	75.7	51.7	84.6	84.0	1%	32%	1%	11%
MON	78.3	52.6	87.4	85.7	1%	28%	1%	9%
LPT	81.9	54.2	88.8	85.0	1%	27%	1%	7%

3.2 Scheduling with Dynamic Job Availability

In dynamic environment the jobs have distinct release times, whereas the machines are still available from the time zero. In this variant the heuristics are coupled with the second meta-algorithm and GP terminal set is expanded according to Table 2. We conducted four sets of experiments, two sets without and another two with sequence dependent setups. For each group we experimented with two different scheduling criteria: weighted tardiness and makespan. All sets consisted of 20 runs, out of which the best evolved priority function is compared with existing heuristics on evaluation set of test cases. For the variant without setup times and with weighted tardiness optimization (notation: $Q\,|r_j|\sum w_j T_j$) we achieved mean value of 35.0 with $\sigma = 1.4$. Additional 20 runs are conducted with makespan as GP fitness function (notation: $Q\,|r_j|\,C_{\max}$), for which the mean value was 42.7 with $\sigma = 0.8$; the results for both sets are shown in Table 3.

It can be seen that GP can easily outperform other heuristics for arbitrary scheduling criteria. On the other hand, it is not very likely that a single heuristic will dominate over more than one criteria, which is particularly true for our GP system with single fitness function. If we are after a heuristic with good overall performance, then it is maybe advisable to take some 'general use' existing heuristic, but if we want to maximize efficiency for a single criteria, then the evolved heuristics represent a good choice.

The last two sets of experiments included setup times, and for the first set we conducted 20 runs with weighted tardiness as fitness function (notation: $Q\,|r_j, s_{ij}|\sum w_j T_j$), for which we achieved mean of 52.9 and $\sigma = 1.7$. Finally, 20 runs were conducted with makespan as the performance criteria (notation: $Q\,|r_j, s_{ij}|\,C_{\max}$), and the obtained mean value was 73.9 with $\sigma = 0.34$. The results for both sets are shown in Table 3.

It can be perceived that in the case of a relatively rare scheduling environment, such as dynamic job arrivals and sequence dependent setups, GP evolved heuristic easily outperforms existing algorithms. This may be attributed to the non-existence of appropriate algorithms for this kind of problem, and that is exactly the situation in which this technique offers the most promising use.

4 Conclusion

This paper shows how genetic programming can be used to build scheduling algorithms for multiple machine environment with arbitrary criteria. The scheduling heuristic is divided in two parts: a meta-algorithm, which is defined manually, and a priority function, which is evolved by GP. We defined the appropriate meta-algorithms for static and dynamic variants of the problem, as well as functional and terminal elements which form the GP solution. The results are promising, as for given problems the evolved solutions exhibit better performance on unseen scheduling instances than existing scheduling methods. Heuristics obtained with GP have shown to be particularly efficient in cases where no adequate algorithms exist, and we believe this approach to be of great use in those situations.

References

1. Jones, A., Rabelo, L.C.: Survey of job shop scheduling techniques. Technical report, NISTIR, National Institute of Standards and Technology, Gaithersburg (1998)
2. Walker, S.S., Brennan, R.W., Norrie, D.H.: Holonic job shop scheduling using a multiagent system. IEEE Intelligent Systems (2) (2005) 50
3. Dimopoulos, C., Zalzala, A.: A genetic programming heuristic for the one-machine total tardiness problem. In: Proceedings of the Congress on Evolutionary Computation. Volume 3. (1999)
4. Dimopoulos, C., Zalzala, A.M.S.: Investigating the use of genetic programming for a classic one-machine scheduling problem. Advances in Engineering Software **32**(6) (2001) 489
5. Adams, T.P.: Creation of simple, deadline, and priority scheduling algorithms using genetic programming. In: Genetic Algorithms and Genetic Programming at Stanford 2002. (2002)
6. Yin, W.J., Liu, M., Wu, C.: Learning single-machine scheduling heuristics subject to machine breakdowns with genetic programming. In: Proceedings of the 2003 Congress on Evolutionary Computation CEC2003, IEEE Press (2003) 1050
7. Atlan, B.L., Polack, J.: Learning distributed reactive strategies by genetic programming for the general job shop problem. In: Proceedings 7th annual Florida Artificial Intelligence Research Symposium, IEEE, IEEE Press (1994)
8. Miyashita, K.: Job-shop scheduling with gp. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000), Morgan Kaufmann (2000) 505
9. Cheng, V., Crawford, L., Menon, P.: Air traffic control using genetic search techniques. In: IEEE International Conference on Control Applications, Hawai'i, IEEE (1999)
10. Hansen, J.V.: Genetic search methods in air traffic control. Computers and Operations Research **31**(3) (2004) 445
11. Lee, Y.H., Bhaskaran, K., Pinedo, M.: A heuristic to minimize the total weighted tardiness with sequence-dependent setups. IIE Transactions **29** (1997) 45–52
12. Lee, S.M., Asllani, A.A.: Job scheduling with dual criteria and sequence-dependent setups: mathematical versus genetic programming. Omega **32**(2) (2004) 145–153
13. Morton, T.E., Pentico, D.W.: Heuristic Scheduling Systems. John Wiley & Sons, Inc. (1993)
14. Jakobovic, D., Budin, L.: Dynamic scheduling with genetic programming. Lecture Notes in Computer Science **3905** (2005) 73
15. Pinedo, M.: Offline deterministic scheduling, stochastic scheduling, and online deterministic scheduling: A comparative overview. In Leung, J.Y.T., ed.: Handbook of Scheduling. Chapman & Hall/CRC (2004)
16. Mohan, R., Rachamadugu, V., Morton, T.E.: Myopic heuristics for the weighted tardiness problem on identical parallel machines. Technical report, The Robotics Institute, Carnegie-Mellon University (1983)
17. Feldman, A., Pinedo, M., Chao, X., Leung, J.: Lekin, flexible job shop scheduling system. http://www.stern.nyu.edu/om/software/lekin/ (2003)

Group-Foraging with Particle Swarms and Genetic Programming

Cecilia Di Chio[1] and Paolo Di Chio[2]

[1] Department of Computer Science,
University of Essex, UK
cdichi@essex.ac.uk
[2] Dipartimento di Sistemi e Istituzioni per l'Economia,
University of L'Aquila, Italy
pdc@ec.univaq.it

Abstract. This paper has been inspired by two quite different works in the field of Particle Swarm theory. Its main aims are to obtain particle swarm equations via genetic programming which perform better than hand-designed ones on the group-foraging problem, and to provide insight into behavioural ecology. With this work, we want to start a new research direction: the use of genetic programming together with particle swarm algorithms in the simulation of problems in behavioural ecology.

1 Introduction

This paper merges and extends ideas from two separate lines of research in Particle Swarm (PS) theory [4].

Firstly, in [7], Poli et al. investigate novel extensions to the classic Particle Swarm Optimisation (PSO) algorithms, using Genetic Programming (GP) to automatically evolve the equations governing the particles in a PS system. This therefore contributes to an important new research trend: using search algorithms to discover new search algorithms. This process has two equally important facets:

- it provides an automated method to evolve better optimisation algorithms;
- it encourages the discovery of emergent, de-centralised solutions (where the overall behaviour is not easily predicted from the constituent parts) by minimising the impact of design constraints stemming (consciously or not) from human design preferences.

The second point means that well-performing solutions can be analysed after the event and, hopefully, generalised to wider problem spaces.

Secondly, in [2], Di Chio et al. provide an innovative example of how the PS paradigm can be applied as a simulation tool for a typical problem in behavioural ecology: group-foraging. They argue that the standard PS algorithm contains elements which map closely to the group-foraging problem (namely social attraction and communication between individuals), and it is therefore surprising

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 331–340, 2007.
© Springer-Verlag Berlin Heidelberg 2007

that it has remained largely a technique used in classical optimisation problems. Some minor changes to the basic PS algorithm allow it to be used as a simplified model of abstract animals, with the problem function representing a foraging environment (with food as optima). Even with such a simple model, qualitatively realistic behaviour was observed, with the emergence of group-foraging behaviour amongst the particles by means of the PS algorithm.

This paper borrows ideas from [7] to extend the approach in [2], by evolving the adapted PS equations via GP. The constraints applied to the GP (in terms of the terminal set, etc.) can be used to reflect restrictions on the behavioural "features" of the animals (and thus reflect more real world cases as required). Running the simulations on simplified foraging landscapes should also help alleviate one of the difficulties with [7]: for the classic mathematical problem functions used, the GP evolved equations which were often very complex, making it difficult to interpret them in terms of the physics of the particles. The main aim is to obtain PS equations via GP which perform better than the standard and hand-designed ones on the group-foraging problem, and which, on analysis, provide insight into behavioural ecology contexts.

The rest of the paper is structured as follows. In section 2 we briefly describe the biological model introduced in [2]. In section 3 we introduce the GP and how this fit in the model. Section 4 presents the results, both for the GP (sec. 4.1) and for the simulation (sec. 4.2), together with their analysis. We conclude in section 5.

2 Food Particle Swarm (FPS) Model

The simulation of animal grouping behaviours is a fairly complex task. In [2], the authors focused on an abstraction of the group-foraging problem, with the following simplified scenario:

- there are neither predators nor other sources of risk or danger;
- the population size stays constant;
- animals can neither see nor smell the food;
- animals can communicate with each other regardless of the dimension of the world;
- the food neither deteriorates nor regenerates.

Animals, represented by particles, move in a flat landscape scattered with food patches. When one animal lands on a patch, it stops on it to eat. When the food finishes, the animal starts its search for a new patch of food. The amount of food eaten by each animal/particle is its fitness (eq. 1), which can be interpreted as the energy gained when feeding. When a particle stops feeding and starts looking for new food, its fitness decreases.

$$fit_i(t) = \begin{cases} fit_i(t-1) + EnG & \text{if particle is on patch and food is available} \\ fit_i(t-1) * EnC & \text{otherwise} \end{cases} \quad (1)$$

where EnG is the energy gained by eating and is equal to the amount of food eaten, and EnC is the energy consumed by moving (in this simple model, is equal to a random negative number).

Since the particles are driven by PS equations (eq. 2-4), they attract each other according to their fitness. Therefore, when the first particle reaches a patch, its fitness will increase and this will attract other particles on the same patch.

$$f_i = \phi_1 R_1 (x_{s_i} - x_i) + \phi_2 R_2 (x_{p_i} - x_i) \tag{2}$$

with the first component (*social interaction*) causing the particles to be attracted with random magnitude to the best position found by the swarm, and the second (*individual learning*) causing the particles to be attracted with random magnitude to the best position found by themselves.

$$v_i(t) = \begin{cases} 0 & \text{if food is on patch} \\ Random & \text{if food is just finished} \\ k((\omega v_i(t-1)) + \Delta t f_i) & \text{otherwise} \end{cases} \tag{3}$$

which is different from the standard PS velocity update equation since it allows the particles to stop on the food patches; when a particle leaves the patch, its velocity is re-initialised at random.

$$x_i(t) = x_i(t-1) + \Delta t v_i(t) \tag{4}$$

where Δt is a factor used to decrease the step the particles take when they move, and has been introduced to promote a smoother movement and more refined search.

3 GP Approach

GP is used to evolve programs in the form of trees. Following the idea first presented in [7], we want to use the function f of equation 2 as the program to be evolved. Our aim is to evolve PS force updating equations that outperform the standard one on the group-foraging problem as described in section 2.

Since we want to evolve force functions which will allow the particles to eat as much food as possible, we have used two different performance measures for the GP:

total fitness - bearing in mind that the fitness of a particle is the amount of food it eats, for the first performance measure we want to maximise the average of the total swarm fitness, which is given by the sum of each particle's fitness

$$fit_s = \sum_{i=1}^{N} fit_i$$

residual food - for the second performance measure, we have instead used the criteria of minimising the total amount of food left on the patches at the end of the simulation.

In order to evolve PS equations which will be able to perform well on different scenarios, we have run the GP on three different food landscape configurations:

1. a single large source, covering most of the landscape, with a large amount of food;
2. three medium sources of different sizes and with different food amounts, covering a restricted portion of the landscape;
3. ten small sources, sparsely scattered on the landscape, all with the same amount of food.

We have used PSs with 10 particles, and run them for 50 iterations on each landscape configuration. Since the position of the patches, as well as the initial position of the particles, can influence the behaviour of the simulation, we have trained the GP on each one of these three scenarios for 8 times with different random number seeds. Table 1 summarises the settings for the GP.

Table 1. Koza style table for the evolution of PS force equations

Objective	Evolve $f_i(x_i, x_{s_i}, x_{p_i})$ over 8 instances of 3 different landscape configurations
Terminal set	Particle position x_i, particle best x_p, swarm best x_s, 0.5, 1.5
Function set	ADD, SUB, MUL, INV
Fitness cases	1. Total swarm fitness 2. Total amount of food left at the end of simulation
Raw fitness	Koza fitness
Population size	1024
Initialisation method	Half builder
Simulation time	8 GP generations, each for 50 iterations
Crossover probability	0.9
Mutation probability	0.0
Initial program length	Half-half ramp
Selection scheme	Tournament selection
Termination criteria	Number of generations (simulation time)

4 Results

4.1 GP Results

From the GP phase, we have obtained the following 6 evolved force functions (3 landscape configurations $\times$ 2 fitness cases).

1. **FPSGP1** One food patch, total fitness

$$f_i = x_i + x_{p_i}$$

For this scenario, the behaviour of the particles is completely selfish. The force acting on the particles makes them move in a direction between their

current position and their own previous best. We can therefore infer that, when there is only one single patch with a large amount of food (we can assume that there is enough food for each particle), and the criteria to select the equation is based on the population fitness, the particles prefer to avoid having a social behaviour.

2. **FPSGP2** Three food patches, total fitness

$$f_i = x_{s_i} + 2x_{p_i}$$

In this scenario, the force acting on the particles is directed somewhere in between the swarm best found position and their personal best, with a fairly strong bias towards the latter point. The current position does not have any influence on the behaviour of the particles.

3. **FPSGP3** Ten food patches, total fitness

$$f_i = 0.5\,(x_i + 0.5x_{s_i} + 0.5x_{p_i})$$

In this case, the force acting on the particle directs them to a point which is amongst their current position, their previous best position and the swarm best position, with a mild bias towards their current location. It seems that, again, the particles tend to give more importance to their own current position than to the others (see case 1), even if in this configuration we cannot really talk of "selfishness", as a partial contribution from the swarm is still present.

4. **FPSGP4** One food patch, residual food

$$f_i = -0.75\,(0.5x_{s_i} + x_{p_i})$$

This scenario is again with only one patch of food, but the fitness criteria is based on the overall amount of food left in the landscape. Even if the swarm component is present, still the influence of the global term is relatively small compared to the influence of the individual one. The particles in this configuration move in a direction which lies between the swarm best and the particle best, with a fairly strong bias towards the particle best.

5. **FPSGP5** Three food patches, residual food

$$f_i = -\,(0.67x_i + 2.33x_{s_i} + 1.75x_{p_i})$$

In this situation, again the influence of the current position on the motion of the particles is limited. With the present configuration, the particles are directed towards a position amongst their current position, their previous best and the swarm best, with a fairly strong bias towards the swarm best.

6. **FPSGP6** Ten food patches, residual food, 100 PS iterations[1]

$$f_i = 0.16x_i + x_{s_i} - 2.16x_{p_i}$$

[1] For this case we have used slightly different GP parameters since the GP settings used for the other experiments gave a very poorly performing force equation.

Despite the fact that this equation contains all three ingredients of the force, the influence of the current particle position is very small. The force acting on the particles tends to move them in a direction parallel to that connecting the particle best position to the swarm best position.

4.2 Simulation Results

To test the evolved PS equations, and compare their performances with those of the FPS algorithm (see eq. 2, sec. 2), we have run a set of simulations, with settings as summarised in table 2.

Table 2. Experiment settings

Runs	20
Iterations	200
Population size	10 particles
Number of patches	1, 3, 10
Patch size	3, from 2 to 3, 1
Food amount	10, from 9 to 10, 5

The performance criteria for the simulation experiments mirrors those used for the selection of forces in the GP phase (see sec. 3). In particular:

- $SumFit$ is the total fitness of the swarm, averaged over the 20 runs, and is given by the sum of the fitness of each particle $\sum_{part} fitness$;
- $MaxSum = \max_{iter}(SumFit)$ is the maximum of the total fitness of the swarm over the 200 iterations;
- $ResFood\#$ is the amount of food left on each patch, with # number of the patch;
- $TotFood$ is the total amount of residual food in the whole landscape, averaged over the 20 runs, and is given by the sum of the residual food of each patch $\sum_{patch} ResFood$;

In tables 3 and 4, we only report the results for the last iteration.

From a first glance at the results, it appears that the FPS performing best is **FPSGP4**, in which the particles are subject to the following evolved force:

$$f_i = -0.75\,(0.5x_{s_i} + x_{p_i}) \quad (5)$$

Particles driven by equation 5 are moved towards a position between the swarm best and the particle best, with a stronger bias towards the particle best. A possible explanation for this behaviour lies in the landscape configuration. Equation 5 has been evolved by the GP on a landscape with a single patch with a large amount of food; therefore, if there is enough food, particles do not need to rely on the information of other members of the swarm to find resources. As a counter-example, we can consider **FPSGP5** (see case 5). This equation has been evolved on a landscape with few patches, each with a medium amount of

Table 3. Total swarm fitness - comparison between hand-designed FPS and GP evolved FPS equations

patches	1		3		10	
	SumFit	*MaxSum*	*sumFit*	*MaxSum*	*sumFit*	*MaxSum*
FPS	7.3839	87.4987	29.5808	210.2540	160.8087	181.2672
FPSGP1	30.5694	74.9876	**119.9199**	177.3456	92.4320	100.9783
FPSGP2	30.5851	79.0632	98.0167	176.7462	93.7175	100.9721
FPSGP3	25.9488	78.9408	101.6207	194.9740	117.8726	127.5567
FPSGP4	21.9781	**91.5746**	105.5064	**218.2384**	**174.0535**	**185.3125**
FPSGP5	**30.7636**	87.3739	88.4477	201.4408	171.9206	178.0254
FPSGP6	4.31189	89.0372	82.0799	215.1310	160.0384	170.0278

Table 4. Total residual food - comparison between hand-designed FPS and GP evolved FPS equations

patches	1		3				10
	TotFood	*ResFood0*	*TotFood*	*ResFood0*	*ResFood1*	*ResFood2*	*TotFood*
FSP	1.6097	1.6097	4.0149	1.1415	1.9504	0.9230	30.2235
FPSGP1	2.9259	2.9259	9.5275	5.4635	2.4923	1.5716	40.2170
FPSGP2	2.4740	2.4740	9.4161	4.4574	3.4225	1.5362	40.1099
FPSGP3	2.4817	2.4817	7.9468	4.1271	2.2835	1.5362	37.5670
FPSGP4	**1.0816**	1.0816	4.6061	3.0908	0.8366	0.6787	**30.8711**
FPSGP5	1.5676	1.5676	6.4731	4.6668	0.7062	1.1000	32.2864
FPSGP6	1.4631	1.4631	**3.3649**	1.3330	1.4986	0.5333	32.7562

food; it seems that, when the food is not enough, or is more difficult to find, the particles prefer to trust the experience of others. There are, in nature, examples of this kind of behaviour [5].

Being evolved on a landscape with only one patch (and residual food performance criteria), it is quite surprising that **FPSGP4** performs well on landscape configurations with more patches. One possible reason is that the absence of the current position term means that, as the search progresses, the particles will tend to move faster and faster, all in the same direction (eventually, the particle best and the swarm best will become the same, and the force will be a constant value), until they find a patch. It seems that, in a landscape scattered with many food patches, a systematic search where all the particles move in parallel is better than one where their movements are less coordinated. Figure 1 shows the trajectories of the particles for both standard and evolved equations. For **FPSGP4** we present the particles' movement over the three different landscape configurations.

Another goal of this work is to observe the emergence of group-foraging behaviour. This means that the particles not only have to eat as much food as they can find, but they also have to do it in groups, i.e., they have to form clusters while feeding. We define a cluster of particles as follows.

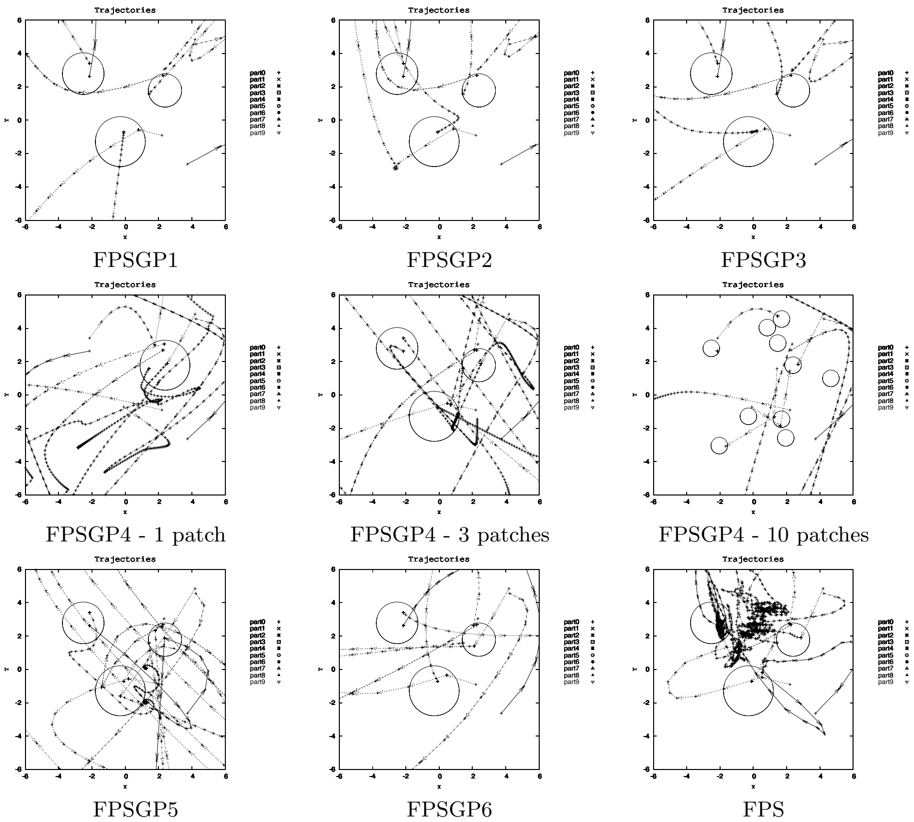


Fig. 1. Particles' trajectories for the different solutions

Definition 1. *Two particles* p_1 *and* p_2 *are in the same cluster if there exists an ordered set of particles* $\{p_{(0,1)}, p_{(0,2)}, \cdots, p_{(0,n)}\}$*, with* $p_{(0,1)} = p_1$ *and* $p_{(0,n)} = p_2$*, such that* $d\left(p_{(0,k)}, p_{(0,k+1)}\right) \leq r$*, where* r *is the cluster threshold.*

Table 5 compares the clustering efficiency of particles acting under the FPS force and those driven by the GP evolved forces (results are for the last iteration only). In particular:

- $\#Clust$ is the number of cluster, averaged over the 20 runs, for each choice of the threshold;
- $\#Part$ is the number of particles in each cluster, averaged over the 20 runs.

As is clear from the results, all the evolved FPSs have a very low clustering. We believe that the landscape configuration plays a role in the efficiency of the particles forming groups. When there is only one patch covering most of the landscape, and the amount of food available is large enough, then the particles will not gather together.

Table 5. Clustering efficiency - comparison between hand-designed FPS and GP evolved FPS equations

patches	1					
threshold	0.5		1.0		1.5	
	$\#Clust$	$\#Part$	$\#Clust$	$\#Part$	$\#Clust$	$\#Part$
FPS	7.15	1.5143	4.8	2.6089	3.5	3.65
FPSGP1	10.0	1.0	10.0	1.0	10.0	1.0
FPSGP2	10.0	1.0	10.0	1.0	10.0	1.0
FPSGP3	10.0	1.0	10.0	1.0	10.0	1.0
FPSGP4	10.0	1.0	10.0	1.0	10.0	1.0
FPSGP5	9.9	1.0111	9.6	1.0472	9.35	1.0778
FPSGP6	10.0	1.0	9.95	1.0055	9.95	1.0055
patches	3					
threshold	0.5		1.0		1.5	
	$\#Clust$	$\#Part$	$\#Clust$	$\#Part$	$\#Clust$	$\#Part$
FPS	7.05	1.5772	4.3	3.2381	2.85	4.45
FPSGP1	10.0	1.0	9.95	1.0055	9.85	1.0167
FPSGP2	9.95	1.0055	9.9	1.0111	9.9	1.0111
FPSGP3	10.0	1.0	9.9	1.0111	9.9	1.0111
FPSGP4	9.95	1.0055	9.8	1.0222	9.6	1.0444
FPSGP5	9.75	1.0292	9.3	1.0847	8.95	1.1270
FPSGP6	9.95	1.0055	9.85	1.0167	9.85	1.0167
patches	10					
threshold	0.5		1.0		1.5	
	$\#Clust$	$\#Part$	$\#Clust$	$\#Part$	$\#Clust$	$\#Part$
FPS	7.9	1.3165	6.2	1.9141	4.1	3.0798
FPSGP1	9.95	1.0055	9.85	1.0167	9.75	1.0278
FPSGP2	10.0	1.0	10.0	1.0	9.85	1.0167
FPSGP3	9.95	1.0055	9.85	1.0167	9.75	1.0292
FPSGP4	9.45	1.0686	8.9	1.1359	8.7	1.16567
FPSGP5	9.6	1.04583	9.1	1.1103	8.9	1.1387
FPSGP6	9.75	1.0292	9.35	1.075	9.1	1.1103

This mirrors what happens in nature, where if enough food is available, animals will maximise their foraging gains by feeding solitarily [3]. Group-foraging efficiency is a trade-off between competing priorities [6]. Theoretical models predict that, while joining a group will not increase an individual ability to find food, the time spent to obtain food is reduced. The trade-off for the reduction in searching time is that a smaller share of food will be available, the only exception being when the resource is so abundant that consumption by one individual does not decrease the availability for others [1].

Another explanation for this poor performance is the topological structure of the particles in the swarm. In this model, we have used a global topology neighbourhood. This means that each particle is able to exchange information with every other particle, therefore reducing the need for particles to group.

5 Conclusion

With this work, we wanted to start a new research direction: the use of genetic programming together with the particle swarm algorithm in the simulation of behavioural ecology problems.

We have shown how, with the aid of GP, it is possible to evolve force equation for the PS which are able to solve group-foraging problems considerably better than the PS algorithm specifically designed for that problem.

There are many future extensions to this work. First of all, we understand that the model presented is a very simple abstraction of a real animal system. Therefore, the first improvement will consist in making the model more biologically plausible. Not only can the FPS model be improved, but the GP can also be made more accurate, incorporating information about the landscape and the animals in the terminals set, other than options and selections in the functions set.

References

1. J. Alcock, *Animal Behavior*, Sinauer Associates, 1998
2. C. Di Chio, R. Poli and P. Di Chio, *Extending the Particle Swarm Algorithm to Model Animal Foraging Behaviour*, ANTS2006 - Fifth International Workshop on Ant Colony Optimization and Swarm Intelligence, 2006.
3. L.A. Giraldeau and T. Caraco, *Social Foraging Theory*, Princeton University Press, 2000.
4. J. Kennedy and R.C. Eberhart, *Swarm Intelligence*, Morgan Kaufmann Publishers, 2001.
5. J. Krause and G.D. Ruxton, *Living in Groups*, Oxford University Press, 2002.
6. D. MacFarland, *Animal behaviour*, Longman, 1999.
7. R. Poli,W.B. Langdon and O. Holland, *Extending Particle Swarm Optimisation via Genetic Programming*, EuroGP, 291-300, 2005.

Multiple Interactive Outputs in a Single Tree: An Empirical Investigation

Edgar Galván-López[1] and Katya Rodríguez-Vázquez[2]

[1] University of Essex, Colchester, CO4 3SQ, UK
egalva@essex.ac.uk
[2] IIMAS-UNAM, Circuito Escolar s/n, Ciudad Universitaria
Del. Coyoacán, México, D.F. 04510, Mexico
katya@uxdea4.iimas.unam.mx

Abstract. This paper describes Multiple Interactive Outputs in a Single Tree (MIOST), a new form of Genetic Programming (GP). Our approach is based on two ideas. Firstly, we have taken inspiration from graph-GP representations. With this idea we decided to explore the possibility of representing programs as graphs with oriented links. Secondly, our individuals could have more than one output. This idea was inspired on the divide and conquer principle, a program is decomposed in subprograms, and so, we are expecting to make the original problem easier by breaking down a problem into two or more sub-problems. To verify the effectiveness of our approach, we have used several evolvable hardware problems of different complexity taken from the literature. Our results indicate that our approach has a better overall performance in terms of consistency to reach feasible solutions.

Kerwords: Multiple Interactive Outputs in a Single Tree, Genetic Programming, Graph-GP representations.

1 Introduction

Genetic Programming (GP) [9] is a heuristic search technique, which has its inspiration from the theories of genetic inheritance and natural selection. This technique has been proved to be a suitable tool for solving problems in many applications. Usually, in GP programs are expressed as syntax trees. However, this form of GP has some limitations. So, some researchers have proposed different type of representations of GP.

For example, Koza [10] proposed Automatically Defined Functions (ADFs). ADF is a function that is dynamically evolved during a run of a GP. The problem with this approach is discovering good ADFs. So, in order to discover if an ADF is good, GP has to spend computation time to discover with which parameters the ADF can be used properly.

Angeline and Pollack [3] proposed a method called Evolutionary Module Acquisition (EMA). The idea of this method is to build and evolve modules (which are the reuse of code) during the evolution process. Because there is not a general

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 341–350, 2007.
© Springer-Verlag Berlin Heidelberg 2007

method of identifying what portions of the individual should be compressed, the composition of each module is selected randomly. The same authors extended this work in [2]. The authors refer to the method as Genetic Library Builder (GLiB).

Montana [13] proposed Strongly Typed Genetic Programming (STGP). He started from the definition of closure (which means that all elements take arguments of a single data type and return values of the same data type). The main characteristic of STGP is to build an individual as a parse tree and the data type of the nodes not necessarily should be the same type.

Teller and Veloso [15] were one of the first researchers to use a graph-based GP. Their method, Parallel Algorithm Discovery and Orchestration (PADO), is a combination of GP and linear discriminator which was used to obtain a parallel classification programs for signals and images.

Poli [14] proposed an approach called Parallel Distributed Genetic Programming (PDGP). Poli stated that PDGP can be considered as a generalisation of GP. However, PDGP can use more complex representations and evolve finite state automata, neural networks and more. PDGP is based on a graph-like representation for parallel programs which is manipulated by crossover and mutation operators and guarantee the syntactic correctness of the offsprings.

Angeline [1] proposed a representation called Multiple Interacting Programs (MIPs). This representation is a generalization of a recurrent neural network that can model any type of dynamic system. Each program in a given set is unique and stored in the form of a parse tree. Using this technique an individual is virtually equivalent to a neural network where the computation performed at each unit is replaced with an independent evolved equation.

Miller [12] proposed Cartesian Genetic Programming (CGP). This technique was called Cartesian in the sense that the method considers a grid of nodes that are addressed in a Cartesian coordinate system. In CGP the genotype is represented as a list of integers that are mapped to directed graphs rather than trees.

Kantschik and Banzhaf [6] proposed a different representation of GP named Linear-Tree. The main idea was to give flexibility to a program to choose different execution paths for different inputs. In this method each program is represented as a tree. Later on, the same authors proposed a representation called Linear-Graph [7]. They argued that graphs come one step nearer to the control flow of a hand written program.

As it can be seen, different ideas have raised in GP to make it more efficient. The aims of this study are: (a) to study how GP behaves with and without the presence of graph-like structures, (b) to incorporate multiple outputs in individual's structures, and, (c) to combine both properties (graph-like structures and multiple outputs in individual's structures) and to see what the effects are.

The paper is organised as follows. In the next section we describe our approach. Section 3 provides details on the experimental setup used and presents results. In section 4 we analyse the results found by our approach and in Section 5 we draw some conclusions.

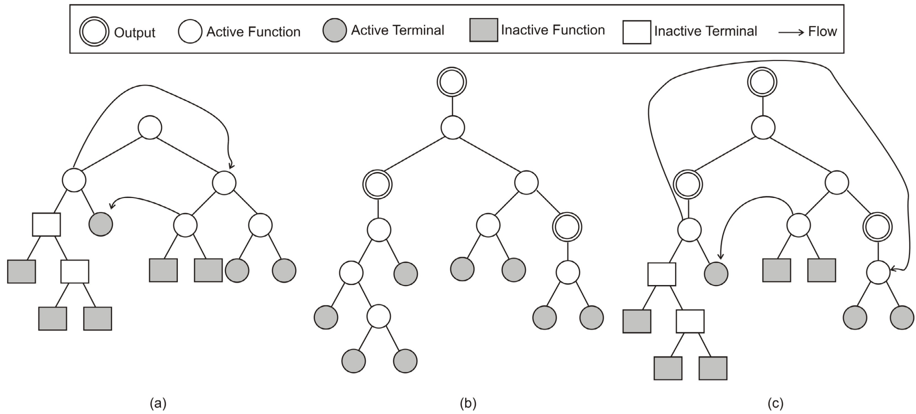


Fig. 1. (a) GP-based representation of graphs without multiple outputs, (b) multiple outputs in a single tree (MOST) without links ,and, (c) multiple interactive outputs in a single tree (MIOST)

2 Approach

Our approach, which we have denominated *Multiple Interactive Outputs in a Single Tree* (MIOST), is based on two ideas. Firstly, we have taken inspiration from graph-GP representations. With this idea we decided to explore the possibility of representing programs as graphs with oriented links, see Figure 1(a). The idea was to replace a function node by other element that represents links which determine what's need to be evaluated. We hope in this way to find parts of an individual that can be more useful in other part(s) of the same individual. Secondly, in our approach a program is represented as a tree as suggested by Koza with the main difference that a program could has more than one output, see Figure 1(b). Let us call this *Multiple Outputs in a Single Tree* (MOST). This idea was inspired on the Divide and Conquer principle, a program is decomposed in subprograms, and so, we are expecting to make the original problem easier by breaking down a problem into two or more sub-problems. In Figure 1(c), we can see a typical individual created with MIOST, which is the result of combining both ideas (graph-gp representation and MOST).

2.1 Output Set

Apart of considering function and terminal sets, as usual, we also consider another set which contain outputs. So, when we create an individual, we choose randomly from the set of outputs any of these and we eliminate it from this set. Once we have created an individual, we check if it contains all the outputs, if not we repeat the process until we have created a valid individual. It is worth mentioning that when we create an individual we do not specify which output will be in the root, so our approach has the power to place the most complex output in the root (results confirm this statement).

2.2 *P* Symbol

The method proposed in this paper does not only allow having more than one output in a single tree but it also allows evolving graph-like structures. This is the result of using the p function symbol, which works as follows:

- Once the individuals in the populations have been generated with its corresponding outputs, we use a probability to replace a function with a p symbol which is a function of arity 2 (This is not a restriction because p could be of any arity).
- If an individual contains this p symbol, this will point to code somewhere in the program, so when p is executed, the subtree rooted at that node is ignored.
- If p symbol points to a function symbol, the p symbol effectively represents the sub-tree rooted at that function.
- If p symbol points to a terminal symbol, the p symbol simply represents that node.

2.3 Genetic Operators

The crossover operator used in MIOST works as usual but an important difference is that, if the sub-tree swapped contained a p symbol, the p symbol's pointer is not changed[1]. Another difference is that once we have created our initial population, we classify each node of each individual in order to know which nodes can be used to apply crossover. With this we assure that an individual will contain the number of outputs that must contain. Of course, this classification is only applied when the individual has more than one output. The mutation operator is applied as usual on a per node basis. The only restriction is that a p symbol is not allowed to be mutated.

2.4 Fitness Function

To test the effectiveness of our approach, we have used several evolvable hardware problems of different complexity taken from the literature. The fitness function works in two stages: at the beginning of the search, the fitness of a genotype is the number of correct output bits (raw fitness). Once the fitness has reached the maximum number of correct outputs bits, we try to optimize the circuits by giving a higher fitness to an individual with shorter encodings.

2.5 Features

The approach detailed above has interesting features. For instance, the presence of p symbols in our representation, assure us that there are inactive code in the

[1] There is an exception to this rule: we prevent a p symbol from referring to a sub-tree that contains the same p since this would lead to an infinite loop. We do this by reassigning the position to which the p in question is pointing to.

Table 1. Truth table of the second example

A	B	C	D	O_1	O_2
0	0	0	0	1	0
0	0	0	1	1	0
0	0	1	0	1	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	1
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	1
1	1	1	1	0	1

Table 2. Truth table of the third example

A	B	C	D	E	O_1	O_2	O_3		A	B	C	D	E	O_1	O_2	O_3
0	0	0	0	0	1	1	0		1	0	0	0	0	0	1	0
0	0	0	0	1	1	0	0		1	0	0	0	1	0	0	0
0	0	0	1	0	1	1	0		1	0	0	1	0	0	1	0
0	0	0	1	1	1	0	0		1	0	0	1	1	0	0	0
0	0	1	0	0	1	1	1		1	0	1	0	0	0	1	1
0	0	1	0	1	1	0	1		1	0	1	0	1	0	0	1
0	0	1	1	0	1	1	0		1	0	1	1	0	0	1	0
0	0	1	1	1	1	1	0		1	0	1	1	1	0	1	0
0	1	0	0	0	1	1	0		1	1	0	0	0	0	1	0
0	1	0	0	1	1	0	0		1	1	0	0	1	0	0	0
0	1	0	1	0	1	1	0		1	1	0	1	0	0	1	0
0	1	0	1	1	1	0	0		1	1	0	1	1	0	0	0
0	1	1	0	0	1	1	1		1	1	1	0	0	1	1	1
0	1	1	0	1	1	0	1		1	1	1	0	1	1	0	1
0	1	1	1	0	1	1	0		1	1	1	1	0	1	1	0
0	1	1	1	1	1	1	0		1	1	1	1	1	1	1	0

individual. This has two advantages: when a mutation takes place in inactive code there is no need to evaluate an individual since there is a change at genotype level but not at phenotype level, and, it allows to study neutrality [8] which is an area of controversial debate on Evolutionary Computation (EC) systems.

3 Experiments

As stated earlier, our approach is based on two ideas. So, for the first problem we will test the first idea. That is, we have used only the graph-like presentation (Figure 1(a)). We used only mutation operator and we have defined different mutation and p rates. For this example, we used the following set of gates $\{AND, OR, NOT\}$.

As a consequence of the results obtained from the first example, we have decided to test MIOST on three different hardware problems with different degrees of complexity. Our results were compared with those obtained by MGA [4], EAPSO [16], EBPSO [11], BPSO [11], EGP [5] and MOST. For these examples, we used the following set of gates {AND, OR, XOR, NOT}. After a series of preliminary experiments we have decided to use crossover rate of 0.7%, mutation rate of 0.02%, and p rate of 0.08% for all examples except for example 1 where we have defined different values. To make a fair comparison with the previous methods, we used the same number of generations and population size.

Runs were stopped when the maximum number of generations was reached. For all examples, we performed 20 independent runs.

Table 3. Truth table of the fourth example

A	B	C	D	O_1	O_2	O_3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

3.1 Example 1

For our first example, we have used the well-known 6-bit Multiplexer Boolean function to verify the idea of graph-like representation (without multiple outputs).

For our first example we have used Population Size (PS) = 200, and Maximum Number of Generations (MNG) = 400, $p = \{0.01, 0.02, 0.03\}$ and mut = {0.02, 0.03, 0.04}.

In Table 4 we can see the results found by the gp-graph representation. In this table we present the percentage of feasible circuits (success rate) and the average number of generations that were necessary to reach the feasible zone[2].

As can be seen, regardless the value of p, the best results were found when mutation rate = 0.02. Does this mean that the presence of p is useless on the evolutionary process? To answer this question, we remove p from the evolutionary search and keep the same mutation rates. The success rates for mutation rates 0.02, 0.03 and 0.4 were 65%, 75%, 45%, respectively.

From these results, it seems to be that the addition of p symbols to the individuals' structures aid the evolutionary search.

3.2 Example 2

For our second example we have used the truth table shown in Table 1. The parameters used in this example are the following: PS = 380 and the MNG = 525 (i.e., a total of 199,500 fitness function evaluations). The same values parameters were used by EGP and MOST. BPSO, EAPSO and EBPSO performed 200,000

[2] The feasible zone is the area of the search space containing circuits that match all the outputs of the problem's truth table.

Table 4. Results found using the gp-graph representation on the 6-bits Multiplexer problem. P and mutation rates are shown in the first and second row, respectively. Feasible Circuits (Success Rate) and Average of Generations (this refers to the average number of generations that were necessary to reach the feasible zone) are shown in the last two rows, respectively.

	% p = 0.01			*% p = 0.02*			*% p = 0.03*		
	%mut=			%mut=			%mut=		
	0.02	0.03	0.04	0.02	0.03	0.04	0.02	0.03	0.04
% of feasible circuits	100%	90%	70%	75%	65%	50%	85%	85%	45%
Average of generations	80.6	155.66	138.62	91.46	136.66	141.5	122.7	109.64	179.5

Table 5. Comparison of results between BPSO, EAPSO, EBPSO, MGA, EGP, MOST and MIOST on the second example

	Feasible circuits	*Avg. # of gates*	*Avg. of gen.*
BPSO	95%	10.05	-
EAPSO	70%	13.45	-
EBPSO	100%	7.75	-
MGA	75%	13.4	-
EGP	55%	9.7	122.9
MOST	85%	14.98	53.29
MIOST	100%	12.9	109.55

fitness function evaluations, while MGA performed 201,300. As we can see in Table 5, the only algorithms able to converge to the feasible region in 100% of the runs were EBPSO and MIOST.

3.3 Example 3

For our third example we have used the truth table shown in Table 2 (Notice that this truth table was split in two due to space limitations). The parameters used in this example are the following: PS = 1,200 and the MNG = 832 (i.e., a total of 998,400 fitness function evaluations). The same values parameters were

Table 6. Comparison of results between BPSO, EAPSO, EBPSO, MGA, GP, EGP, MOST and MIOST on the third example

	Feasible circuits	*Avg. # of gates*	*Avg. of gen.*
BPSO	25%	23.95	-
EAPSO	50%	18.65	-
EBPSO	45%	20.1	-
MGA	65%	17.05	-
EGP	60%	9.66	149.5
MOST	50%	11.3	94.5
MIOST	75%	11.6	104.67

Table 7. Comparison of results between BPSO, EAPSO, EBPSO, MGA, GP, EGP, MOST and MIOST on the fourth example

	Feasible circuits	*Avg. # of gates*	*Avg. of gen.*
BPSO	-	-	-
EAPSO	-	-	-
EBPSO	-	-	-
MGA	-	-	-
EGP	30%	-	-
MOST	10%	20	234.16
MIOST	35%	22.16	277.363

used by EGP and by MOST. BPSO, EAPSO and EBPSO performed 1,000,000 fitness function evaluations, while MGA performed 1,101,040. As we can see in Table 6, MIOST is the algorithm which has the highest percentage of feasible solutions reached (75%).

3.4 Example 4

For our fourth and last example (also know as Katz circuit) we have used the truth table shown in Table 3. The parameters used in this example are the following: PS = 880 and the MNG = 4,000 (i.e., a total of 3,520,000 fitness function evaluations). The same values parameters were used by EGP and MOST. As we can see in Table 7, MIOST is the algorithm which has the highest percentage of feasible solutions reached (35%).

4 Analysis

For analysis purposes, we have conducted our experiments in the following way:

1. Firstly, we have allowed to the individuals having p symbols in their representation and in this way, we have been able to have a gp-graph representation,
2. Secondly, we have used MOST which is form of GP that has allowed us to define multiple outputs in each individual and
3. Thirdly, we have used MIOST, which is a combination of the previous ones. In other words, the individuals in MIOST have p symbols in their structures and multiple outputs.

Let us focus our attention on the first example. In this example we decided to test an individual partially created with our approach. That is, we have specified to build an individual with links without multiple outputs. To verify how good this idea is, we used only mutation operator in this example. The presence of p symbols in individuals' structures is necessary to improve performance on the evolutionary process (results confirm this statement). However, when p was not present in any of our individuals, the performance of the GP was worst.

From the results obtained in the first example, we decided to incorporate this idea to the representation where an individual can have more than one output (MOST). This combination of ideas gave us as a result MIOST representation and it was tested on the last three examples which are more complex problems.

Our results indicate that MIOST has a better overall performance in terms of consistency to reach feasible solutions in all the examples. Let us analyse the results we have found with our approach for the last three examples. For the example 2, the average number of generations to solve the problem using MIOST is 109.55, while in MOST the average number of generations is 53.29. Similar situation is observed in examples 3 and 4, where the average number of generations to solve the problem using MIOST is 104.67 and 277.363, while in MOST the average number of generations is 94.5 and 234.16, respectively. As can be seen, the presence of p symbols in MIOST seems to require more generations to find the feasible circuit. This can be explained, if we consider that mutations could take place on inactive code that does not produce any change at phenotype level. On the other hand, this inactive code can have a role where partial solutions are protected against disrupted mutations. However, further analysis needs to be done to give final conclusions.

5 Conclusions

In this paper we have presented MIOST which is a new form of genetic programming which has two main features: (a) it allows to represent programs as graphs with oriented links (graph-GP representation) and (b) a program can have more than one output.

We have used four evolvable hardware problems of different complexity to carry out our experiments with the proposed approach. Firstly, we used MOST to see how good the idea was by using only multiple outputs in our individuals. Once we saw that this gave us good results, we tested our approach (MIOST) which incorporates the idea of gp-graph representation in the presence of p symbols in individuals' structure.

Our results indicate that MIOST has a better overall performance in terms of consistency in reaching feasible solutions. However, our approach was not able to improve previously published results in terms of number of gates. This is due two reasons (a) our approach is not an optimization technique and (b) our approach has the restriction that one or more outputs depend on the solution of one or more outputs. This can be seen easily by analyzing Figure 1(c).

Acknowledgments

The first author thanks to CONACyT for support to pursue graduate studies at University of Essex. The second author gratefully acknowledges support from CONACyT through project 40602-A. The authors would like to thank the anonymous reviewers for their valuable comments.

References

1. P. J. Angeline. Multiple interacting programs: A representation for evolving complex behaviors. *Cybernetics and Systems*, 29(8):779–806, Nov. 1998.
2. P. J. Angeline and J. B. Pollack. Coevolving high-level representations. July Technical report 92-PA-COEVOLVE, Laboratory for Artificial Intelligence. The Ohio State University, 1993.
3. P. J. Angeline and J. B. Pollack. Evolutionary module acquisition. In D. Fogel and W. Atmar, editors, *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 154–163, La Jolla, CA, USA, 25-26 Feb. 1993.
4. C. Coello and A. Aguirre. Design of combinational logic circuits through and evolutionary multiobjective optimization approach. In *Artificial Intelligence for Engineering, Design, Analysis and Manufacture*, volume 16, pages 39–53, January. 2002.
5. E. Galvan-Lopez, R. Poli, and C. C. Coello. Reusing code in genetic programming. In M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa, and T. Soule, editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 359–368, Coimbra, Portugal, 5-7 Apr. 2004. Springer-Verlag.
6. W. Kantschik and W. Banzhaf. Linear-tree GP and its comparison with other GP structures. In J. F. M. *et al.*, editor, *Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 302–312, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.
7. W. Kantschik and W. Banzhaf. Linear-graph GP—A new GP structure. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 83–92, Kinsale, Ireland, 3-5 Apr. 2002. Springer-Verlag.
8. M. Kimura. Evolutionary rate at the molecular level. In *Nature*, volume 217, pages 624–626, 1968.
9. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, Cambridge, Massachusetts, 1992.
10. J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* The MIT Press, Cambridge, Massachusetts, 1994.
11. E. H. Luna, C. C. Coello, and A. H. Aguirre. On the use of a population-based particle swarm optimizer to design combinational logic circuits. In R. S. Z. *et al.*, editor, *Proceedings of the 2004 NASA/DoD Conference on Evolvable Hardware*, pages 183–190, Los Alamitos, California, June 2004. IEEE Computer Society.
12. J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.
13. D. J. Montana. Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman, Cambridge, MA 02138, USA, 7 May 1993.
14. R. Poli. Parallel distributed genetic programming. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimisation.* McGraw-Hill, 1999.
15. A. Teller and M. Veloso. PADO: Learning tree structured algorithms for orchestration into an object recognition system. Technical Report CMU-CS-95-101, Department of CS, Carnegie Mellon University, Pittsburgh, PA, USA, 1995.
16. X. Xu, R. C. Eberhart, and Y. Shi. Swarm intelligence for permutation optimization: A case study on n-queens problem. In *Proceedings of the IEEE Swarm Intelligence Symposium 2003*, pages 243–246, Indianapolis, Indiana, USA, 2003.

Parsimony Doesn't Mean Simplicity: Genetic Programming for Inductive Inference on Noisy Data

Ivanoe De Falco[1], Antonio Della Cioppa[2], Domenico Maisto[3], Umberto Scafuri[1], and Ernesto Tarantino[1]

[1] Institute of High Performance Computing and Networking, National Research Council of Italy (ICAR–CNR)
Via P. Castellino 111, 80131 Naples, Italy
{ivanoe.defalco,ernesto.tarantino,umberto.scafuri}@na.icar.cnr.it
[2] Natural Computation Lab – DIIIE, University of Salerno, Via Ponte don Melillo 1, 84084 Fisciano (SA), Italy
adellacioppa@unisa.it
[3] University of Modena and Reggio Emilia, Via Campi 111, 41100 Modena, Italy
maisto.domenico@unimore.it

Abstract. A Genetic Programming algorithm based on Solomonoff's probabilistic induction is designed and used to face an Inductive Inference task, i.e., symbolic regression. To this aim, some test functions are dressed with increasing levels of noise and the algorithm is employed to denoise the resulting function and recover the starting functions. Then, the algorithm is compared against a classical parsimony–based GP. The results shows the superiority of the Solomonoff–based approach.

1 Introduction

Inductive Inference is a fundamental problem both in science and engineering. Its aim is to find a functional model of a system by determining its fundamental properties from its observed behavior. In early Sixties, Solomonoff proposed a formal theory for Inductive Inference [1]. He discovered that the Inductive Inference problem can be faced by looking for a program v (a functional expression) among all the programs u which generate x (the string of symbols on a given alphabet encoding the observed data of a given phenomenon) as output, with the highest *a priori* probability, i.e., the one with the shortest length. Unfortunately, an effective procedure able to find the program v does not exist [2]. As a consequence, the effective use of Solomonoff's Induction theory is constrained by Computability Theory. Nonetheless, in practice we can approach Inductive Inference by adopting heuristic procedures.

Genetic Programming (GP) [3] is well suited to face this problem [4,5]. In fact, GP should perform a search for functional expressions that fit all the data, while minimizing their length. Besides, to overcome the possible lack of closure

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 351–360, 2007.
© Springer-Verlag Berlin Heidelberg 2007

property GP is subject to, and, at the same time, to introduce bias into the evolutionary process, Whigham's approach is used by exploiting a GP based on Context–Free Grammars (CFGs) [6]. In [7], in order to drive the evolution, it is adopted a fitness function with a penalty term depending on the number of nodes of the particular tree associated to the expression under evaluation. In fact, intuitively, the higher the number of nodes of a tree the lower the *a priori* probability of generating it according to the chosen CFG.

More recently it has been shown in [2,8] that to solve induction problems by Inductive Inference, it is possible to introduce in a GP scheme, by premising some hypotheses to the model, a fitness function directly derived from the expression of the probability distribution of any possible result for the future experiments as a function of those recorded in the previous n.

This work deals with a comparison between the innovative GP method based on Solomonoff probabilistic induction theory [8,9] and a parsimony GP–based approach on a typical problem of Inductive Inference on an unordered set, i.e., symbolic regression. Namely, we aim to find out which of them can better recover, starting from empirical data, a common property by filtering out the "inconvenience" of random noise, inevitable in scientific measurements.

The paper is organized as follows: Section 2 illustrates an overview on the GP and on the fitness functions used. Section 3 outlines the applications of the different methods to four test functions and their robustness is verified at four levels of noise. Conclusions are eventually left to Section 4. The appendix gives details about Solomonoff's view on induction and hypothesis made to obtain a fitness function for the evolutionary inductive inference approach.

2 General GP Framework

The two inductive systems have a common GP algorithm and differ in the choice of the fitness functions only. Our GP is based on an expression generator providing the starting population with a set of programs different in terms of size, shape and functionality. Such a generator is implemented by means of a CFG ensuring the syntactic correctness of the programs. The genotypes are functional expressions encoded as derivation trees of the adopted CFG. This encoding is appealing since the actions performed by the genetic operators can be implemented as simple operations on the derivation trees. In fact, crossover acts by randomly choosing a nonterminal node in the first parent and then by randomly selecting the same nonterminal node in the second one. Finally, it swaps the related subtrees. If a corresponding nonterminal node cannot be found in the second parent, the crossover has no effect. Mutation works by randomly choosing a nonterminal node and then the corresponding production rule is activated to generate a new subtree, thus resulting either in a complete substitution (macro–mutation) or in variation of a leaf node (micro–mutation).

We now describe the fitness functions used in this comparison. Let us put forward some notations: p is the program to evaluate, σ is the standard deviation

of the empirical data, $\mathcal{Q}$ is the set of the *questions* Q_i from which Inductive Inference is made and n is their number, $p(Q_i)$ is the output of the program for the i–th *question*, A_i is the *answer* related to Q_i. Finally $\omega, w \in]0,1[$ are positive parameters to be chosen in order to ensure a balance between the error made by p on data and a penalty term related to its 'complexity'.

2.1 Solomonoff–Based Fitness Function

Consequently to Solomonoff's theory [2,8] we have defined a GP, named as Solomonoff–based Fitness Algorithm (SFA), which adopts as fitness function:

$$F(p) = \frac{1}{n}\frac{1}{\sigma^2}\sum_{Q_i \in \mathcal{Q}} |p(Q_i) - A_i|^2 - \omega \ln(a_0(p)) \tag{1}$$

where $a_0(p)$ is the *a priori* probability of the program p. Equation (1) is a functional expression of two terms, exactly matching those composing eq. (16) in the Appendix. In fact, the first term is a Mean Square Error (MSE), and evaluates the error made by the program p in the approximation of the problem under examination. The second term, instead, depends on the *a priori* probability of the program p examined. To evaluate it, starting from the CFG, an algorithm computes the *a priori* probability $a_0(p)$ of the derivation tree which generates the expression p.

Computation of the *a priori* probability. Drawing inspiration from [2], the computation of the a_0s is carried out by means of the "Laplace's rule" for successive and independent events. Once specified the grammar, the probability α_{i_p} that a given production rule is present in the i–th node of the derivation tree of the program p is k/m, where k is the number of times in which the production has previously occurred in the tree plus one, and m is the total number of all the productions which are legal there, incremented by the number of times in which they have previously occurred in the definition of the tree. The product

$$a_0(p) = \prod_{i_p=1}^{q} \alpha_{i_p} \tag{2}$$

yields the *a priori* probability of the program p, i.e. of the fruit of the tree with q productions, related to the grammar chosen [8]. This procedure is in accordance with the results obtained by the theory of algorithmic probability, in the sense that it attributes higher probabilities to functions which have lower descriptive complexities with reference to the chosen grammar. So, between two different solutions, the GP will choose the "simplest" expression, thus causing an effective application of "Occam's razor". It should be noted that expressions containing a lower number of primitive functions have lower complexity than shorter expressions using a higher number of primitive functions (see [2]).

2.2 Fitness Function with a Parsimony Term

The fitness is the sum of two weighted terms. The former accounts for the difference between computed and actual answers on $\mathcal{Q}$, while the latter is proportional to the number of nodes of the derivation tree:

$$F(p) = \frac{1}{n}\frac{1}{\sigma^2}\sum_{Q_i \in \mathcal{Q}} |p\,(Q_i) - A_i|^2 + wN_p \tag{3}$$

where N_p is the number of nodes making up the derivation tree for p. The aim of the parsimony term is to allow that, during the first phase of expression discovery, the error term is predominant. So the algorithm is free to increase the tree size, either in depth or in width, in such a way to ease the search towards the exact match of the data. Then, the system will exploit the obtained solutions to achieve shorter and shorter expressions. We have decided to name the system using eq. (3) as Parsimony–based Fitness Algorithm (PFA).

2.3 Linking Parsimony and Solomonoff

By adopting a general framework that links PFA and SFA fitness directly, it is possible to show that the parsimony term in eq. (3) is a particular case of the Solomonoff's term in eq. (2). In fact, starting from Solomonoff fitness function and by supposing that the grammar production rules have a uniform probability to be applied in a certain node, we can achieve the PFA fitness. In fact, by defining the probability α_{i_p} of a node i_p simply as $1/n$ (where n is the number of grammar production rules), we have:

$$a_0(p) = \prod_{i_p=1}^{q} \alpha_{i_p} = \left(\frac{1}{n}\right)^q \tag{4}$$

where q is the number of nodes and therefore, eq. (1) becomes:

$$F(p) = \frac{1}{n}\frac{1}{\sigma^2}\sum_{Q_i \in \mathcal{Q}} |p\,(Q_i) - A_i|^2 - \omega\, q \ln(1/n)\,. \tag{5}$$

The above shows that eq. (2) is a first order compression of evolved expressions, while eq. (4) is the zero-th order. Then, like it should be in a bayesian inductive approach, the particular way to compute the *a priori* probabilities influences the quality of a prediction. In fact, if one wants to strive toward significant models, particularly in the inductive task, it should be looked for *structural* models. One wishes for structures which, however remotely, do say something about the *causal* mechanism behind the data at hand. Equation (3) does not give any insight in this direction, while eq. (1) contains important features of Solomonoff's universal *a priori* probability distribution that can approximate any probability with extreme accuracy.

Table 1. GP parameters setting

number of data points	200
population size	400
max generations	1000
max depth trees	8
grammar rules	$S \rightarrow f(x) = E$ $E \rightarrow (EOE) \mid F(E) \mid N \mid R \mid x$ $O \rightarrow + \mid - \mid * \mid /$ $F \rightarrow sqr \mid sqrt \mid abs \mid sin \mid cos$ $N \rightarrow c \in \mathbb{N}, R \rightarrow d \in \mathbb{R}$
tournament selection size	10
crossover probability	30%
macro-micro mutation probability	100% (60% in macro and 40% in micro mutation)

3 Experimental Comparison

Experiments have been run on four functions of increasing complexity, i.e., the Quartic polynomial, the Schwefel and the Rastrigin functions, and a version of the Michalewicz function with 5 minima [10], in their 1–dimensional version:

$$f(x) = x^4 + x^3 + x^2 + x, \;\; -1 \le x \le 1 \tag{6}$$

$$f(x) = -x \cdot \sin\left(\sqrt{|x|}\right), \;\; -500 \le x \le 500 \tag{7}$$

$$f(x) = 10 + x^2 - 10 \cdot \cos\left(2\pi x\right), \;\; -5.12 \le x \le 5.12 \tag{8}$$

$$f(x) = -\sin(x) \cdot \sin^2\left((5/\pi)x^2\right), \;\; 0 \le x \le 3.14 \tag{9}$$

This function set has been chosen to test PFA and SFA by problems of incremental difficulty, meaning here that the creation of a particular constant can determine the success or the failure of the symbolic regression. In fact, the quartic polynomial is the simplest considered case and represents a problem without constants, the second problem introduces integer numbers, while the third and the fourth ones need real constants as well.

For both algorithms 20 runs with different random seeds have been carried out with the same parameters resumed in Table 1. Other important parameters are the weights ω and w. Given that both the error term and the order of magnitude of the complexity term are the same in both eq. (1) and eq. (3) as well, it is evident the values of ω and w should be kept equal in order to correctly compare the two complexity measures in the equations. After a preliminary tuning phase, we found that in all the cases examined 10^{-3} is the best value.

The experimental results are included in Fig. 1 and in Table 2.

3.1 Experimental Findings on Test Functions Dressed with Noise

An Inductive Inference system is a useful tool for scientific investigations if it works in spite of the inevitable random noise which is present in all real measures.

As a consequence, we wish to examine the ability of the two aforementioned algorithms to understand the underlying function even in the presence of noise. Hence we dress the functions with known noise. In this way we can check performance of the two GP systems as a function of the level of noise, and we might compare them against those achieved in absence of noise. Starting from a target function $f(x)$ we define a noisy function g by adding noise as follows:

$$g(x) = f(x) + \xi(x) \tag{10}$$

where $\xi(x)$ is a Gaussian pseudo–random number distribution with zero average.

We accomplish our investigation by characterizing any noisy data series g in terms of the percent relative level of noise:

$$\eta_f(g) = (||f - g||/||g||)^2 \cdot 100 \tag{11}$$

where $|| \cdot ||$ is the norm in L_2.
We have taken into account four different values for η_f, namely 5%, 10%, 15% and 20%. For any level each original function has been dressed with 4 noise data series differing one another, because provided by a noise generator with different random seeds, but all having a value around the prefixed η_f noise level. In fact, we wish to examine the algorithms' robustness in extracting the function independently of the particular values of the noise series added. This is accomplished by computing the Normalized Root Mean Square Error (NRMSE) of the best individual found at the end of each run with respect to f:

$$\text{NRMSE} = 100 \cdot \frac{\sqrt{\frac{n}{n-1} \cdot \text{MSE}}}{\sigma_t} \tag{12}$$

where n is the number of training data, and σ_t the standard deviation of the target values. This measure assigns the value of about 100 to an expression that performs equivalently with the mean target value, while a perfect fit is obtained when the NRMSE reaches 0. For any given noisy series we have then run any GP algorithm 20 times, so as to investigate average behavior from the evolutionary point of view. As a result, for any value of η_f each algorithm has been executed 80 times. Fig. 1 shows grafically the relative performance of both PFA and SFA in terms of average NRMSE with its standard error when the algorithms converge to the original functions by "filtering" out the noise. As it can be seen, PFA meets increasing difficulty in solving the task as the level of noise increases. SFA, instead, shows at all the levels of noise a better ability to find the original function evidenced by lower values of both average NRMSE and standard error. Of course, such ability decreases as the level of noise becomes higher.

Another interesting topic deals with the denoising capacity of the two algorithms. The idea is that the noise is the part of data which cannot be compressed with a considered model, while the remaining is the information beared by the signal. Such a denoising capacity cannot be evidenced by taking into account only the results outlined in Fig. 1. To this aim, for each function and for each noise level considered, we computed the average NRMSE of the noisy

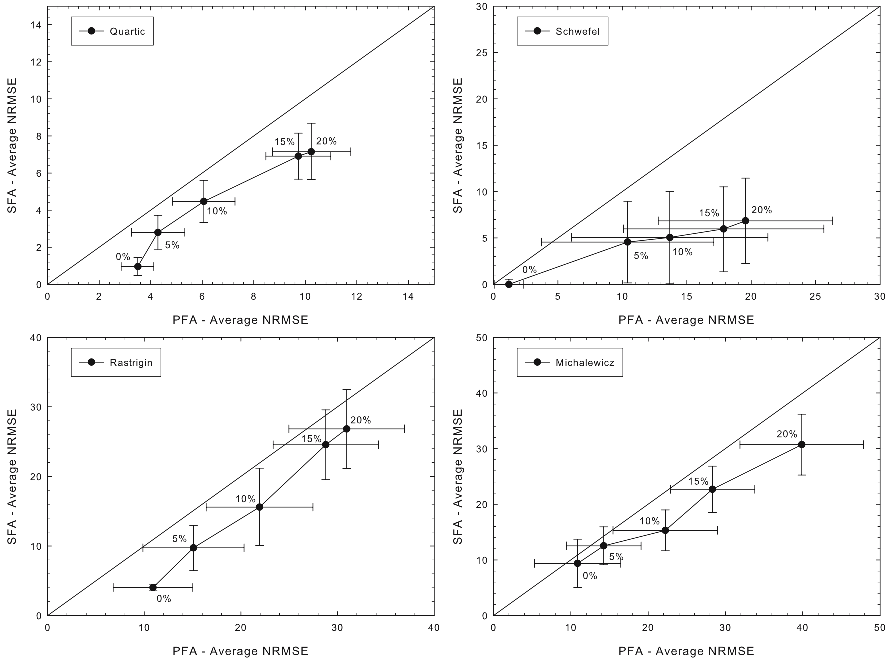


Fig. 1. Performance of SFA with respect to PFA in terms of the average NRMSE and its standard error

Table 2. Results in terms of average NRMSE

		Average NRMSE							
		Quartic		Schwefel		Rastrigin		Michalewicz	
Noise level	Noised Data	PFA	SFA	PFA	SFA	PFA	SFA	PFA	SFA
0%	–	3.50	0.96	1.20	0.00	10.91	4.03	10.88	9.37
5%	$\sim$ 23	4.28	2.80	10.41	4.56	15.10	9.74	14.25	12.53
10%	$\sim$ 34	6.06	4.47	13.68	5.06	21.94	15.58	22.22	15.31
15%	$\sim$ 40	9.73	6.91	17.86	5.97	28.78	24.55	28.31	22.69
20%	$\sim$ 48	10.23	7.15	19.56	6.85	30.95	26.83	39.88	30.71

data g with respect to the original function f. Such averages are reported in the second column of Table 2 along with those of the evolved solutions. It should be noted that the average NRMSE is about the same for all the considered functions. The experimental findings confirm that both PFA and SFA reduce overfitting in that the average NRMSE of the evolved solutions is lower than the average NRMSE of the noisy data (see Table 2). However, overfitting on noisy values takes place more frequently for PFA. On the contrary, SFA avoids more frequently overfitting, thanks to the introduction of the *a priori* probabilities.

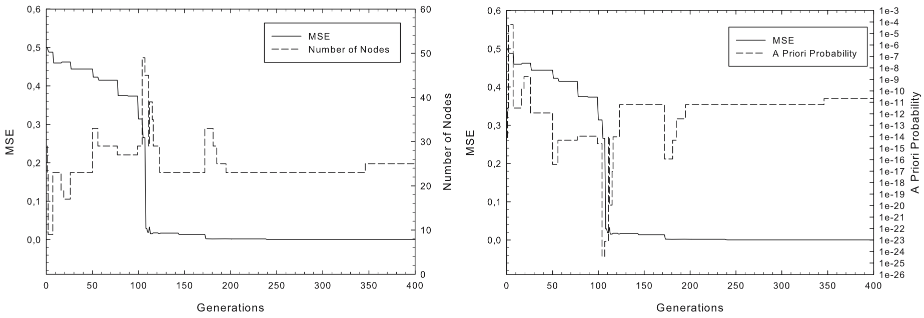


Fig. 2. Behavior of MSE and number of nodes (left) and of the *a priori* probability (right) as a function of time t, for a typical SFA run on the Rastrigin function

Parsimony doesn't mean simplicity. The whole of results underlines that a search procedure using a fitness with a parsimony term, which looks for a solution optimized with respect to a resource (in this case the number of nodes), is not capable of finding the individuals with the simplest expression, meaning here, in the common sense, the most 'regular', i.e., the one with the largest probability of being generated. Moreover, the same results indicate that a search procedure which uses a fitness function based on a formal notion of 'simplicity' is more sensitive than a parsimony procedure. This is evident, for example, for Quartic and Schwefel functions in which, even at 20%, in many runs SFA was able to discover the original functions, while although the best individuals found by PFA have in many runs NRMSE values with respect to the g data lower than those shown by SFA, they are not able to catch the underlying function f. This is due to the parsimony term which leads to prefer, at parity of number of nodes, expressions which better fit the g data. In other words, it can distinguish between two functions having the same number of nodes and the same performance on the basis of the way in which those trees have been generated (the number of rules used, the number of times each rule is called upon, etc.).

As a confirmation, Fig. 2 (left) reports a typical evolution of SFA in terms of MSE and number of nodes on the Rastrigin function, while Fig. 2 (right) shows the variations in the *a priori* probability for the best individuals found during that run. It is evident that once reached a sufficiently low MSE value, and as long as it remains unchanged, the algorithm tries to find individuals whose expressions have larger *a priori* probability, i.e., it tries to find a simpler expression for the same function. Moreover, the most important feature of SFA is the following: until about the 350-th generation the best solution found is an individual with a derivation tree made of 23 nodes, whose expression is $10+sqr(x)-10*\cos(6.28*x)$. At that time the system discovers another solution whose expression is $10+(x*x)-10*\cos(6.28*x)$ in which $sqr(x)$ is replaced by $(x*x)$, the number of nodes is 25, the MSE is the same as in the previous case, but one function less than the previous one is needed. For PFA the fitness of this latter solution is higher than the former, while for SFA the fitness decreases.

As a consequence, PFA could never select it as the best solution of the run, because the parsimony term is proportional to the number of nodes.

4 Conclusions and Future Work

In this paper two GP systems are used to perform symbolic regression, one based on parsimony ideas (PFA) and the other on Solomonoff probability induction concepts (SFA). They both perform quite well when functions without noise are faced. Strong differences arise concerning the application of the two systems on data with additive gaussian noise, as soon as the level of noise increases. In fact, experimental data show that PFA is able to recover the original functions for quite low noise levels only. SFA, instead, shows a better ability in discovering exactly the original functions, thus further reducing overfitting occurrence, although, of course, its performance decreases when the noise level gets larger. This behavior allows SFA to achieve a better NRMSE with respect to PFA when computed on the original functions at all levels of noise.

Starting from the results obtained (see eq. (5) for example), in the future work we plan to deepen the formal links between the algorithmic *a priori* probability and evolutionary learning.

References

1. Solomonoff, R.J.: A formal theory of inductive inference. Information and Control **7**(1) (1964) 1–22, 224–254
2. Solomonoff, R.J.: Progress in incremental machine learning. In: NIPS Workshop on Universal Learning Algorithms and Optimal Search, Whistler, B.C. (2002)
3. Koza, J.R.: Genetic Programming: on the programming of computers by means of natural selection. MIT Press, Cambridge, Massachusetts (1992)
4. Zhang, B., Mühlenbein, H.: Balancing accuracy and parsimony in genetic programming. Evolutionary Computation **3**(1) (1995) 17–38
5. Gagné, C., Schoenauer, M., Parizeau, M., Tomassini, M.: Genetic programming, validation sets, and parsimony pressure. In Collet, P., Tomassini, M., Ebner, M., Gustafson, S., Ekárt, A., eds.: Lecture Notes in Computer Science. Volume 3905., Springer-Verlag (2006) 98–109
6. Whigham, P.A.: Grammatical Bias for Evolutionary Learning. PhD thesis, School of Computer Science, University of New South Wales, Australia (1996)
7. De Falco, I., Della Cioppa, A., Passaro, A., Tarantino, E.: Genetic programming for inductive inference of chaotic series. In: Lecture Notes in Artificial Intelligence. Volume 3849., Springer-Verlag Berlin Heidelberg (2005) 156–163
8. De Falco, I., Della Cioppa, A., Maisto, D., Tarantino, E.: A genetic programming approach to Solomonoff's probabilistic induction. In: Lecture Notes in Computer Science. Volume 3905., Springer-Verlag Berlin Heidelberg (2006) 24–35
9. Solomonoff, R.: Private communications. (2006)
10. Gordon, V.S., D. Whitley, D.: Serial and parallel genetic algorithms as function optimizers. In Forrest, S., ed.: Proceedings of the Fifth International Conference on Genetic Algorithms, San Mateo, CA, Morgan Kaufman (1993) 177–183

A From Solomonoff's Theory to a Fitness Function

According to Solomonoff, a generic induction problem on an unordered set can be solved by defining an operator able to satisfy the following request: let $[Q_i, A_i]$ be an unordered set of n couples (Q_i, A_i), $i = 1 \ldots n$, where Q_i and A_i can be strings and/or numbers. Given a new element Q_{n+1}, which is the function F such that $F(Q_i) = A_i$, for $i = 1, \ldots, n+1$, and with the highest *a priori* probability?

From bayesian considerations and by supposing that each couple (Q_i, A_i) is independent of the others, assigned a new Q_{n+1}, the corresponding distribution probability for any possible A_{n+1}, as a function of the data set $[Q_i, A_i]$, is [2]:

$$\mathrm{P}(A_{n+1}) = \sum_j a_n^j \ \mathrm{O}^j(A_{n+1}|Q_{n+1}), \qquad \text{with} \qquad a_n^j = a_0^j \prod_{i=1}^{n} \mathrm{O}^j(A_i|Q_i)\,, \tag{13}$$

where $\mathrm{O}^j(\cdot|\cdot)$ is the conditional probability distribution with respect to the function F^j such that $F^j(Q_i) = A_i$, and where a_0^j is the *a priori* probability associated to F^j. Solomonoff called eq. (13) **Induction Operator** (IO) [2]. Of course, it is impossible to compute the infinite summation in eq. (13) by using finite resources, yet it can be approximated by using a finite number of terms, i.e. those having a high value of a_n^j. In the ideal case we could include the terms with maximal weights among the a_n^js, but no effective procedure exists able to carry out this search, since some among the u programs might not terminate, but this cannot be known *a priori*. Thus, the application of the IO consists in trying to find, in a preset time, a set of functions F^js such that the summation:

$$\sum_j a_n^j \tag{14}$$

be as high as possible. For deterministic induction, in particular, it is sufficient to find, in a finite time, the term among the a_n^js which dominates eq. (14).

Let us suppose that any A_i is given by $A_i = F^j(Q_i) + \varepsilon_j$, with F^j a deterministic function of Q_i, and ε_j the error of the function F^j on data. We also assume that ε_j has a normal distribution with standard deviation σ independent of both Q_i and j.

By the above hypotheses, the $\mathrm{O}^j(\cdot|\cdot)$s are distributions of gaussian probabilities with standard deviation σ, and hence it results (see [8]):

$$a_n^j = a_0^j \ (2\pi\sigma^2)^{-\frac{n}{2}} \exp\left(-\sum_{i=1}^{n} \frac{[F^j(Q_i) - A_i]^2}{2\sigma^2}\right) \tag{15}$$

Then, rather than maximizing eq. (14), we minimize, with respect to the functions F^js, the negative of the natural logarithm of its terms:

$$-\ln a_n^j \approx \frac{1}{2\sigma^2} \sum_{i=1}^{n} \Big[F^j(Q_i) - A_i\Big]^2 - \ln a_0^j \tag{16}$$

in such a way, a deterministic induction problem on a set $[Q_i, A_i]$ can be solved by finding a F^j with respect to which eq. (16) is as low as possible [8,9].

The Holland Broadcast Language and the Modeling of Biochemical Networks

James Decraene, George G. Mitchell, Barry McMullin, and Ciaran Kelly

Artificial Life Laboratory,
Research Institute for Networks and Communication Engineering,
School of Electronic Engineering, Dublin City University, Ireland
{james.decraene,george.mitchell,mcmullin,ciaran.kelly}@eeng.dcu.ie
www.eeng.dcu.ie/ alife/

Abstract. The Broadcast Language is a programming formalism devised by Holland in 1975, which aims at improving the efficiency of Genetic Algorithms (GAs) during long-term evolution. The key mechanism of the Broadcast Language is to allow GAs to employ an adaptable problem representation. Fixed problem encoding is commonly used by GAs but may limit their performance in particular cases. This paper describes an implementation of the Broadcast Language and its application to modeling biochemical networks. Holland presented the Broadcast Language in his book "Adaptation in Natural and Artificial Systems" where only a description of the language was provided, without any implementation. Our primary motivation for this work was the fact that there is currently no published implementation of the Broadcast Language available. Secondly, no additional examination of the Broadcast Language and its applications can be found in the literature. Holland proposed that the Broadcast Language would be suitable for the modeling of biochemical models. However, he did not support this belief with any experimental work. In this paper, we propose an implementation of the Broadcast Language which is then applied to the modeling of a signal transduction network. We conclude the paper by proposing that with some refinements it will be possible to use the Broadcast Language to evolve biochemical networks *in silico*.

Keywords: Broadcast Language, adaptable representation, biochemical networks modeling.

1 Introduction

Holland proposed the Broadcast Language so as to address some potential limitations in the application or performance of Genetic Algorithms (GAs) [6,3]. Holland argued that GAs provide an efficient method of adaptation; however in the case of long-term adaptation, the efficiency of GAs could be limited by the representation used to encode the problem. In traditional GAs, this representation is fixed and may significantly influence the complexity of the fitness landscape. During long-term evolution this may limit the performance of the GA.

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 361–370, 2007.
© Springer-Verlag Berlin Heidelberg 2007

To overcome this limitation, Holland proposed to dynamically adapt the problem representation. Adapting the representation may then generate correlations between the problem representation and the GA performance.

Another feature discussed by Holland was the conjecture that the Broadcast Language is a Turing Complete programming language. If this is so, it would imply that the language would not dictate any long-term limits to its evolution. However although this issue clearly has intrinsic interest, it will not be consided further in the current paper.

Following this, Holland argued that the Broadcast Language would provide a straightforward representation for a variety of natural models such as Genetic Regulatory Networks or Neural Networks. This would show the computational power of the Broadcast Language and its capacity to adapt.

However, while recognising some of the potential merits of the Broadcast Language, we need to consider the fact that Holland did not support this approach with experimental evaluation; nor have we been able to identify any body of subsequently published work on the Broadcast Language in the literature.

We believe that there is a need for further investigations on the Broadcast Language because:

- The Broadcast Language may provide a useful framework for investigating a range of interesting problems in Evolutionary Computation and Theoretical Biology.
- The potentially interesting applications of the Broadcast Language were only outlined, not actually formally demonstrated, by Holland.
- Since Holland's early presentations [6], no further work on Broadcast Systems (Broadcast Language-based systems) can be found in the literature.

To initiate these further investigations we have implemented an execution platform for the Broadcast Language. We applied this to the study of the modeling of biochemical networks. This paper is organized as follows: we first introduce in more detail the Broadcast Language and then describe our implementation of the Broadcast System. We then demonstrate how to model a signal transduction network with the Broadcast Language. This is finally followed by a discussion of possible refinements toward the modeling of a specific problem instance: the evolution of biochemical networks *in silico*.

2 The Broadcast System

We use the formalism given by Holland in the original text [6]. We initially provide an overview of the Broadcast System and then present our implementation.

2.1 An Overview

The Broadcast Language basic components are called *broadcast units* which are strings formed from the set $\Lambda = \{0,\ 1,\ *,\ :,\ \Diamond,\ \triangledown,\ \blacktriangledown,\ \triangle,\ p,\ '\}$. Broadcast units can be viewed as *condition/action* rules. Whenever a broadcast unit conditional

statement is satisfied, the action statement is executed. This means that whenever a broadcast unit detects, in the environment, the presence of one or more specific signal(s), possibly including the broadcast units themselves, then the broadcast unit would broadcast an output signal.

As an example, we may consider a given broadcast unit that upon detecting signals I_1 and I_2 would broadcast an output signal I_3. This is analogous to a biological phenomenon where an enzyme would form a product upon the binding of specific substrate(s) to its binding region(s). In this example an enzyme can be thought of as a broadcast unit, substrate(s) would be detected signal(s), the enzyme binding region(s) would refer to the broadcast unit condition part, the product is the output signal and finally the environment would be the reaction space (e.g., the cell).

Following the above analogy, a substrate can be degraded during catalysis. We implement this phenomenon through the signal processing ability of broadcast units. Indeed general signal processing can also be performed with broadcast units: e.g., a broadcast unit may detect a signal I and broadcast a signal I', so that I' is some modification of the signal I.

Some broadcast units may broadcast a signal that may constitute a new broadcast unit. Similarly, a broadcast unit can be interpreted as a signal detected by another broadcast unit. As a result, a broadcast unit may create new broadcast units or detect and modify an existing broadcast unit.

A set of broadcast units, combined as a string, is designated a *broadcast device*. A broadcast device can be viewed as analogous to a protein complex in which interactions between the several proteins result in complex functional behavior of the molecule.

Holland also described in detail how he distinguishes between four key *types* of broadcast unit, designated types 1, 2, 3 and 4. See [6,2] for a detailed description of this and also the more general syntax and semantics of the Broadcast Language.

2.2 The System

In this section we present our implementation of the Holland Broadcast System. We have implemented the Broadcast System using an Object Oriented paradigm, in which we may distinguish three main classes:

- `Env` represents the environment, this object holds a list of all current existing devices.
- The class `BDevice` designates a broadcast device, an instantiation of `BDevice` may hold from 0 to n `BUnit` objects.
- The `BUnit` class refers to a broadcast unit, it may contain one or two argument(s) and an output signal, all represented by strings of characters.

In this system based on discrete timesteps, the sequential operation is as follows. At timestep t, all broadcast devices including null devices are stored

in a vector of devices S. This vector is held by an instance of `Env`. A vector of character strings A is used to hold signals (strings) to be added to S at the beginning of t. At time $t = 0$, S is empty and A represents the initial set of broadcast devices. D is a vector of strings holding signals to be removed from S at the end of timestep t.

Figure 1 presents an overview of the system from its initialization to its termination.

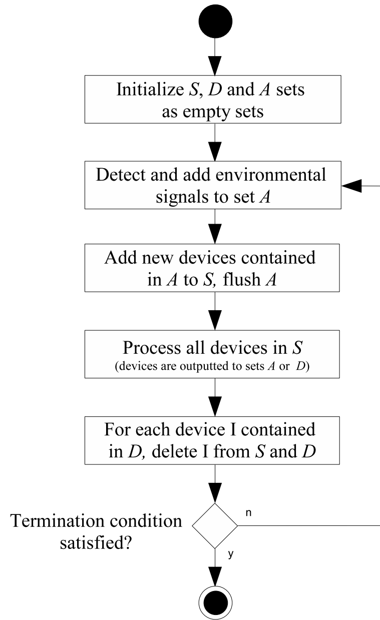


Fig. 1. Broadcast System flowchart

Following this, we discuss in detail each step presented in this diagram:

1. *Initialization*: an `Env` object is instantiated, vectors S, A and D are created and are empty by default.
2. *Environmental signals*: at this step, input signals (strings of character) given by the environment are added to set A. At time $t = 0$, the input signals correspond to the initial set of signals. A *detector* may be built to probe the "external" environment and insert new signals into set A.
3. *Transferring signals from set A to S*: signals contained in set A are inserted in set S. Set A is then flushed. Each signal inserted in S is processed into broadcast devices (`BDevice` objects); if a signal generates an active broadcast device then this broadcast device is parsed into broadcast units (`BUnit` objects).

4. *Processing signals in S*: this step is broken up into two sequential subprocesses:
   (a) we first look for broadcast units of type 4 (see [6]) that are able to broadcast at the same time t. If those broadcast units can be satisfied by other signals (including themselves) then they broadcast their output signals. The latter output signals are then directly inserted into S. As these newly inserted signals may satisfy other similar type 4 broadcast units, it is necessary to repeat this whole process until no new signal gets inserted into S. This is the first subprocess to be performed because type 4 broadcast units may output signals that may contribute to other broadcast units contained in S at time t.
   (b) Then each broadcast device in S is processed in a sequential order: if a broadcast device I is active then each broadcast unit I_i contained in I may broadcast its output signal upon detecting adequate signals. A broadcast unit which has already been activated at time t may not broadcast again within that timestep, under any circumstances. Output signals issued by type $1, 2$ and 4 broadcast units are stored in set A. If a type 2 broadcast unit is activated then its output signal is inserted into set D. Finally, if a broadcast device I is a null device and is not a persistent signal, then this device signal is added to set D.
5. *Delete signals from sets S and D*: for each signal I_d contained in set D, if there is a signal of the form I_d present in S then this signal is deleted from S. If there are n signals in S that are of the form I_d then only one of those signals is deleted (selected uniformly at random). D is then flushed.
6. *Termination condition*: this condition is set by the user, for example it may be an integer T indicating the maximum number of steps to be completed. If this user-defined termination condition is not satisfied then the system returns to step 1.

The above implementation addresses or clarifies a number of ambiguities that had been left open by Holland. We now show how the Broadcast Language is capable of modeling biochemical networks, which was one of Holland identified application areas.

3 Modeling a Biochemical Network

In this section we present a case study where we model a biochemical network with the Broadcast Language. We successfully model a signal transduction network, which was previously modeled with the aid of a Boolean network [9]. Note that this example given by Genoud only addresses the regulatory aspects of the signaling network.

One way to represent the regulatory aspects of a biochemical network is to use the Boolean formalism. With the Boolean abstraction, a (protein) molecule is considered as a logical expression having two different possible states. One possible state is the *on* state meaning that the molecule is present in the environment. To the contrary, when a molecule state is *off*, this indicates that this

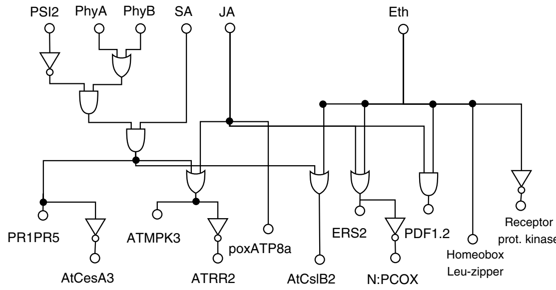


Fig. 2. Boolean representation of the signal transduction network controlling the plants defense response against pathogens

particular molecule is not present in the environment (cell). Figure 2 provides an example of a graphical boolean representation of a signal transduction network.

We use the Broadcast Language to mirror the Boolean network of the biochemical network presented in Figure 2. To accomplish this, we proceed to a direct mapping of each Boolean function to broadcast devices. Using this Broadcast System model, one may determine the states of the output molecules according to the states of the input molecules.

We first represent each molecule (substrate) $PhyA$, $PhyB$, Eth, etc., with a string (signal) such as $p0000000$, $p0000001$, $p0000010$, etc. We then define the broadcast devices (enzymes) which enable the reactions to occur in this network. In this case, the broadcast devices stand for the boolean functions shown in Fig.2.

$$(PR1PR5) = (\neg PSI2 \wedge (PhyA \vee PhyB)) \wedge SA$$

The above equation describes the state of $PR1PR5$ according to the states of $PSI2$, $PhyA$, $PhyB$ and SA. We now present how to express this Boolean expression using the Broadcast Language.

In order to represent an `OR` gate that takes for input signals $PhyA$ and $PhyB$ we generate the following broadcast device:

$$I_1 = *p000000\Diamond : 1000000$$

This broadcast device indicates that whenever persistent signals $p0000000$ or $p0000001$ ($PhyA$ or $PhyB$) are detected, the signaling molecule 1000000 is broadcast. This example also demonstrates how to represent *crosstalk* phenomena in the Broadcast Language. The purpose of using signaling molecules will be shown in the description of the third broadcast device I_3.

The `NOT` gate is expressed through the use of type 2 broadcast unit. To represent `NOT` $p0000010$ ($PSI2$), the following broadcast device is defined:

$$I_2 = * : p0000010 : 1000001$$

The above broadcast device stipulates that when no persistent $PSI2$ molecule is present then the signaling molecule 1000001 is broadcast at time $t + 1$.

Following the given example, we want to express an `AND` gate. The expression (($p0000000$ `OR` $p0000001$) `AND` (`NOT` $p0000010$)) can be translated into the following broadcast device:

$$I_3 = *1000000 : 1000001 : 1000010$$

I_3 would broadcast 1000010 only if 1000000 *and* 1000001 are detected. The detection of 1000000 indicates that either $p0000000$ ($PhyA$) or $p0000001$ ($PhyB$) is on. Secondly, detecting 1000001 implies that $p0000010$ ($PSI2$) has not been detected.

$$I_4 = *p0000011 : 1000011$$

The broadcast device I_4 is used to broadcast a signaling molecule 1000011 if $p0000011$ (SA) is detected.

$$I_5 = *1000010 : 1000011 : 1000100$$

I_5 is similar to I_3 and represents an `AND` gate taking into account the results of I_3 and I_4. This broadcast device, if satisfied, broadcasts a signaling molecule that is employed to activate $PR1PR5$ ($p0000101$), as follows:

$$I_6 = *1000100 : p0000101$$

Case:	1	2	3	4	5
Inputs	[PhyA] off	[PhyA] on	[PhyA] on	[PhyA] off	[PhyA] on
	[PhyB] on	[PhyB] off	[PhyB] off	[PhyB] off	[PhyB] on
	[PSI2] on	[PSI2] on	[PSI2] on	[PSI2] off	[PSI2] on
	[SA] off	[SA] on	[SA] off	[SA] off	[SA] on
	[JA] on	[JA] off	[JA] off	[JA] off	[JA] on
	[ETH] on	[ETH] off	[ETH] on	[ETH] off	[ETH] on
Outputs	[PR1PR5] off	[PR1PR5] off	[PR1PR5] off	[PR1PR5] off	[PR1PR5] off
	[ATMPK3] on	[ATMPK3] off	[ATMPK3] off	[ATMPK3] off	[ATMPK3] on
	[AtCesa3] on	[AtCesa3] on	[AtCesa3] on	[AtCesa3] on	[AtCesa3] on
	[ERS2] on	[ERS2] off	[ERS2] on	[ERS2] off	[ERS2] on
	[N:PCOX] off	[N:PCOX] on	[N:PCOX] off	[N:PCOX] on	[N:PCOX] off
	[poxATP8a] on	[poxATP8a] off	[poxATP8a] off	[poxATP8a] off	[poxATP8a] on
	[ATRR2] off	[ATRR2] on	[ATRR2] on	[ATRR2] on	[ATRR2] off
	[AtCslB2] on	[AtCslB2] off	[AtCslB2] on	[AtCslB2] off	[AtCslB2] on
	[PDF1.2] on	[PDF1.2] off	[PDF1.2] off	[PDF1.2] off	[PDF1.2] on
	[HomLeuZip] on	[HomLeuZip] off	[HomLeuZip] on	[HomLeuZip] off	[HomLeuZip] on
	[RePrkinase] off	[RePrkinase] on	[RePrkinase] off	[RePrkinase] on	[RePrkinase] off

Fig. 3. A series of results obtained with our implementation of the Broadcast System. The Boolean network representation of the signal transduction network (Fig. 2) was implemented with our system. A molecule is *on* when at least one occurrence of the corresponding broadcast device is found after time $t = 4$. These results present the states of the output molecules $PR1PR5$, $AtCesA3$, etc according to the differing states of the input molecules $PhyA$, $PhyB$, etc.

The whole Boolean network may be built following the above described method. This case study was implemented with our system and tested against a selection of inputs, and the outputs reacted precisely in accordance with the boolean functions specified by the network, see Fig. 3. We may note that because some broadcast units broadcast at time $t+1$, a cascade of similar reactions may then take a certain

amount of time steps to process the whole network. This is indeed necessary so that every boolean functions described in the model are processed. In the current example, 4 time steps are necessary to obtain the output states accounting for every boolean gates.

This example showed that the Broadcast Language is a straightforward method to model a biochemical network when the latter is described with a Boolean formalism. The same method could also be applied to represent other genetic regulatory networks as they can be modeled with Boolean networks [7].

4 Discussion

In the case study above, we demonstrated that the Broadcast Language can model Genetic Regulatory Networks (GRNs). The ability of the Broadcast Language to mirror Boolean networks illustrates the wide ranging processing power that Broadcast Systems are capable of.

A key advantage to using the Broadcast System, as mentioned by Holland, is the ability of the system to work in conjunction with GAs. By allowing the coupling of GAs with the Broadcast System, a variety of evolutionary operators (mutation, crossover etc) are accessible. With these operators it would be possible to design Broadcast Systems that model the evolution of GRNs.

Previous works on the modeling of the evolution of GRNs can be found in the literature [5,1]. Nevertheless we believe that the study of the Broadcast Language would complement this understanding. As argued by Holland [6], One benefit is that with an adaptable representation the Broadcast Language would prevent evolutionary plateaus being encountered during the evaluation. In long term evolution, this may be of high significance as we commonly meet such plateaus in evolutionary systems [4,8].

Although the modeling of the evolution of GRNs is valuable, we focus on a related, but currently not so well understood class of biochemical network, the Cell Signaling Networks (CSNs).

In the case study we provided above, we presented a CSN model where only the regulatory aspects of the CSN were covered. Although this qualitative approach is of interest, this significantly limits the power of Broadcast Systems to model biochemical networks. As currently defined, the Broadcast System cannot express concentration kinetics and it is well known that molecular concentrations play an important role in chemical reactions.

In order to refine the Broadcast Language we outline some refinements which focus on the following points:

- To incorporate chemical kinetics in Broadcast Systems.
- To strengthen the biological plausibilities in the modeling of CSNs with Broadcast Systems.
- To facilitate the evaluation of the Broadcast System.

Examining the first point we must consider collision theory: molecules must collide to react together. When the molecular concentration increases, the probability of collision increases as well. These collisions occur at random and are

best described as Brownian motion. However simulating Brownian motion is computationally expensive. We approximate this phenomena in the Broadcast System by adjusting the way broadcast devices are processed:

- Instead of processing all broadcast devices sequentially during a time step, we propose the following: at each time step t, we pick n pairs of broadcast devices at random. For each pair of devices, one of the broadcast devices is designated (at random) as the *catalyst device* and the second one as the *substrate device*. If the conditional statement of the catalyst device is satisfied by the signal of the substrate device, then the action statement of the catalyst device is executed upon the substrate device.
- n number of pairs of broadcast devices is a constant and refers to the temperature in real chemistry. Temperature has an important role in chemical reactions, indeed molecules at higher temperature have a greater probability to collide with one another. In the Broadcast System, in order to increase the "temperature", one may increment the integer number n.

In order to improve the biological application of the system, and to facilitate its evaluation, the following refinements are proposed:

- In the Broadcast Language specification given by Holland, additional rules were required to resolve some ambiguities raised by the interpretation of broadcast devices. To facilitate this, we suggest to simplify the nature of broadcast units by preserving broadcast units of type 1 only.
- Similarly the notion of non-persistent devices is removed: by default all devices are considered as persistent molecules.
- As type 3 broadcast units and non-persistent devices no longer exist in our proposal, no molecule can be deleted from the population. However the deletion of molecules is needed to obtain evolutionary pressure. Our suggestion is as follows: each time two molecules react together, we pick a molecule at random and delete it from the population.

The above suggestions simplify and strengthen the ability of Broadcast Systems to model biochemical networks. However to model precisely real biochemical networks, more attributes are needed to describe accurately these complex systems. A solution is to implement this derivation of the Broadcast System as an agent-based model, where the agents behavior and adaptation is determined by broadcast devices. This allows the definition of additional molecular properties (e.g., spatial location, state, etc) for each agent.

Although our proposed work will require further evaluation to precisely represent real biochemical networks, these refinements allow for the design of an evolutionary simulation platform to study *artificial* biochemical networks *in silico*.

5 Conclusion

In this paper we presented our implementation of the Holland Broadcast System and demonstrated the modeling of a signal transduction network with this approach. This work was motivated by the desire to implement the Holland system

and also to apply it to biochemical networks modeling. We evaluated our implementation and showed that the Broadcast Language is suitable to model GRNs. We then discussed the benefits of Broadcast Systems to evolve GRNs through the use of GAs. Nevertheless it was shown later that Broadcast Systems are limited regarding the study of biochemical networks from a quantitative point of view. Following this, we proposed refinements that allow the Broadcast Language to model the evolution of biochemical networks accounting for the quantitative aspects. These refinements provide the following additional benefits: reinforcement of the biological applications of the system and facilitate its evaluation. This approach may contribute to the understanding of the evolutionary dynamics of biochemical networks.

Acknowledgement

This work was funded by ESIGNET (Evolving Cell Signaling Networks in Silico), an European Integrated Project in the EU FP6 NEST Initiative (contract no. 12789).

References

1. Stefan Bornholdt. Modeling Genetic Networks and Their Evolution: A Complex Dynamical Systems Perspective. *Biol. Chem*, 382(9):1289–1299, 2001.
2. James Decraene. The Holland Broadcast Language. Technical Report ALL-06-01, Artificial Life Lab, RINCE, School of Electronic Engineering, Dublin City University, 2006.
3. David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.
4. Dominique Groß and Barry McMullin. The creation of novelty in artificial chemistries. In *ICAL 2003: Proceedings of the eighth international conference on Artificial life*, pages 400–409, Cambridge, MA, USA, 2003. MIT Press.
5. Jennifer Hallinan and Janet Wiles. Evolving genetic regulatory networks using an artificial genome. *Proceedings of the second conference on Asia-Pacific bioinformatics-Volume 29*, pages 291–296, 2004.
6. John H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.
7. Stuart A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, May 1993.
8. Tim Taylor. Creativity in evolution: individuals, interactions, and environments. pages 79–108, 2002.
9. Marcela B. Trevino Santa Cruz Thierry Genoud and Jean-Pierre Mtraux. Numeric simulation of plant signaling networks. *Plant Physiology*, 126:1430–1437, August 2001.

The Induction of Finite Transducers Using Genetic Programming

Amashini Naidoo[1] and Nelishia Pillay[2]

[1] School of Computer Science, Univesity of KwaZulu-Natal, Westville Campus, Westville, KwaZulu-Natal, South Africa
Amashini.Naidoo@vodacom.co.za
[2] School of Computer Science, University of KwaZulu-Natal, Pietermaritzburg Campus, Pietermartizburg, KwaZulu-Natal, South Africa
pillayn32@ukzn.ac.za

Abstract. This paper reports on the results of a preliminary study conducted to evaluate genetic programming (GP) as a means of evolving finite state transducers. A genetic programming system representing each individual as a directed graph was implemented to evolve Mealy machines. Tournament selection was used to choose parents for the next generation and the reproduction, mutation and crossover operators were applied to the selected parents to create the next generation. The system was tested on six standard Mealy machine problems. The GP system was able to successfully induce solutions to all six problems. Furthermore, the solutions evolved were human-competitive and in all cases the minimal transducer was evolved.

Keywords: genetic programming, finite state transducers.

1 Introduction

Finite state transducers are used in various domains of Computer Science such as natural language processing and image processing. A fair amount of research has investigated the evolution of finite state machines, however almost all of these efforts have been focused on inducing finite acceptors and very little work has examined the effectiveness of evolutionary algorithms in generating finite transducers. The study presented in this paper is an initial attempt at assessing genetic programming (GP) for the purpose of inducing finite transducers, namely, Mealy machines.

The following section provides an overview of finite state transducers. Section **3** presents a brief introduction to genetic programming and describes previous attempts at evolving transducers. A genetic programming system for evolving Mealy machines is proposed in section **4**. Section **5** discusses the overall methodology employed to evaluate GP as means of generating Mealy machines. The performance of this GP system on six standard finite transducer problems is analyzed in section **6**. Finally, section **7** summarizes the findings of this study.

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 371–380, 2007.
© Springer-Verlag Berlin Heidelberg 2007

2 Finite Transducers

Finite transducers are finite state machines that produce an output string for a given input string. There are two types of transducers, namely, Mealy machines and Moore machines. Mealy machines are formally defined as:

$$M_e = \{q_0, \sum, \Gamma, Q, \delta\}$$

where q_0 is the start state
$\sum$ is the input string alphabet
Γ is the output string alphabet
Q is the set of states
δ: Q x $\sum$ x $\Gamma \rightarrow$ Q, i.e. defines the transitions

A Mealy machine that takes a string consisting of *a*'s and *b*'s as input and outputs a binary string containing a *1* at the position of every substring "*ab*" in the original string is illustrated in **Figure 2.1**.

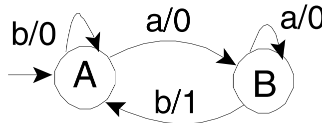


Fig. 2.1. Example of a Mealy machine

Notice that Mealy machines produce the output string as part of the transition. For example, at state *A* if a *b* is read in a *0* is output. Moore machines differ from Mealy machines in that the output string is produced at the state. Both these machines are deterministic. Algorithms exist for converting Mealy machines to Moore machines, thus only the induction of Mealy machines will be examined in this study.

3 GP and Finite Transducers

Genetic programming (GP) is an evolutionary algorithm that represents elements of a population using variable sized structures [1]. Examples of representations used by different genetic programming systems include parse trees, matrices, and directed graphs, just to name a few. GP has been successfully employed in numerous domains and in a number of cases has produced human-competitive results. Research into the generation of finite automata using evolutionary algorithms was initiated as early as the 1960's with the study by Fogel et al. [2] investigating the evolution of deterministic finite automata (DFAs). Later attempts at inducing DFAs include the genetic algorithm implemented by Dupont [3] to evolve DFAs. This system takes a maximal canonical automaton for a set of positive sentences for the language as input.

Dunay et al. [4] employ a genetic programming system to generate S-expressions representing DFAs. Brave [5] takes a similar approach that uses genetic programming to evolve cellular automata encodings of the DFAs. More recent studies investigate

the use of gene regulation [6] and a genetic programming system representing each DFA as a transition matrix [7], for the purposes of DFA evolution. While a lot of work has been directed at examining the generation of finite acceptors, not much research has addressed evolving finite transducers. The only study applying evolutionary algorithms to this domain is that conducted by Lucas [8] to induce chaining codes for binary images.

Lucas [8] implements a random-hill climber to induce a Mealy machine for converting a 4-direction chain code for a binary image to an 8-direction chain code. Each potential solution transducer is a table representing the transition matrix of the corresponding finite transducer. Each transducer is comprised of a maximum of ten states. One of three nondestructive mutation operators is applied at each stage of the search to further evolve the potential solution.

The training set consisted of fifty pairs of input strings and their corresponding target output. A general solution was induced within fifty runs of the system. The solution consisted of five redundant states which once removed yielded a transducer that was equivalent to the target machine. The following section presents a genetic programming system, using a direct representation for each finite transducer, namely, a transition graph, to evolve Mealy machines.

4 Proposed GP System

This section proposes a genetic programming system for the evolution of Mealy machines. The generational control model is employed by the system. The number of elements of the population is kept fixed from one generation to the next. During each generation the reproduction, mutation and crossover operators are applied to parents that have been selected using tournament selection. These processes are described in more detail below.

4.1 Representation

Each element of the population is represented as illustrated in **Figure 2.1**. A direct representation is used for each Mealy machine, i.e. each element of the population is a directed graph with each node representing the state of the finite transducer and each edge representing the transition between states. An edge label specifies an input character and the corresponding output character. The Mealy machines are deterministic, hence each state has an outgoing arc for each element of the input alphabet and a state can have at most one outgoing arc for each element of the input alphabet. Thus, the terminal set is comprised of elements of the input and output alphabet and function nodes represent states in the Mealy machine. During initial population generation each individual is created by randomly choosing source and destination states and input and output characters for the edges joining these nodes, until the maximum node limit per individual is reached.

4.2 Interpretation, Fitness Evaluation and Selection

The fitness cases are essentially pairs of input strings and the corresponding output string that must be produced by the Mealy machine. The interpretation process takes

an input string and the start state of the finite transducer as input. As each character of the input string is processed the corresponding transition of the transducer is applied and the output characters specified by each transition are concatenated to produce the output string. The fitness of an individual is the number of fitness cases for which it produces the correct output string. These fitness measures are used by the tournament selection method to choose the parents for the next generation.

4.3 Genetic Operators

The GP system applies the reproduction, mutation and crossover operators to the selected parent to create the next generation. The reproduction operator basically clones the chosen parent. **Figure 4.3.1** provides an overview of the crossover process.

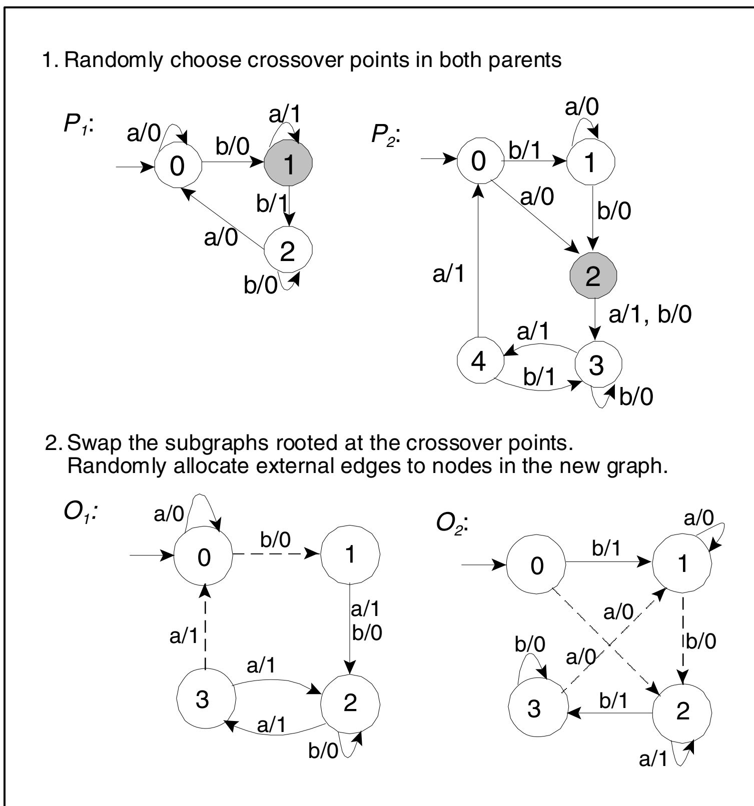


Fig. 4.3.1. Application of the crossover operator

The crossover operator randomly selects crossover points in copies of both the selected parents. The subgraphs rooted at these points are swapped. *Internal* edges refer to those edges directed at nodes remaining in the parent while *external* edges refer to edges that were connected to nodes in the removed subgraph. In **Figure 4.3.1**

internal edges are illustrated by a solid line and external edges by a broken line. The external edges in both parents are randomly allocated to target nodes in the newly inserted subgraphs.

The mutation operator replaces a randomly chosen subgraph in the selected parent with a newly generated subgraph. All external edges, i.e. edges that were previously connected to nodes in the removed subgraph, are randomly redirected to nodes in the new subgraph. This process is illustrated in **Figure 4.3.2**.

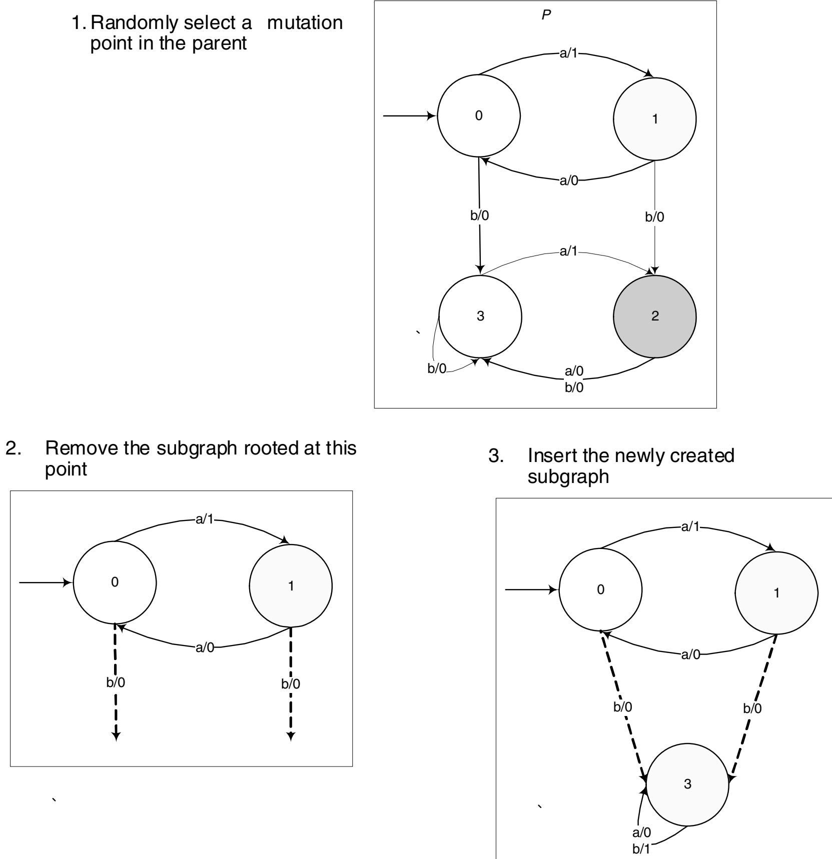


Fig. 4.3.2. Application of the mutation operator

Genetic operators often produce offspring that have worse fitness than their parents.

Unfortunately, in such cases these operators impede the success of the GP algorithm in finding solutions. Thus, non-destructive versions of the both these operators, i.e. genetic operators that produce offspring that are at least as fit as their parents, have been implemented and will be used if needed. In both cases the operation is repeated until an offspring at least as good as the parent (or one of the

parents in the case of crossover) is produced. A limit is set on the number of improvement steps. If this limit is exceeded the offspring with worse fitness than its parent is accepted.

5 Experimental Methodology

The study reported in this paper is an initial attempt at evaluating genetic programming as a means of evolving finite transducers. Hence, the genetic programming system proposed in the previous section will be tested on the standard finite transducer benchmarks described in the general literature on theory of machines and formal languages such as [9]. These benchmarks are listed in **Table 5.1**. Based on the findings of this study, revisions will be made to the original system and further evaluations will be performed in a specific application domain such as natural language processing.

Table 5.1. Mealy machine data set (from [9] and [10])

Machine	Description	Example
M1	Mealy machine that outputs a 1 for each substring 'aaa', Σ = {a, b}.	Input: aaaaabaaa Output: 001000001
M2	Mealy machine that outputs a 1 for each substring 'aab', Σ = {a, b}.	Input: abaabaab Output: 00001001
M3	Mealy machine that takes a binary string in reverse order as input and outputs a binary string representing the number input incremented by 1.	Input: 001 Output: 101
M4	Mealy machine that outputs a 1 at every double letter, Σ = {a, b}.	Input: baaabbabb Output: 001101001
M5	Mealy machine that takes a binary string as input and outputs the 1's complement of the string.	Input: 011010 Output: 100101
M6	Mealy machine that takes a binary string as input and outputs a string consisting of E's and O's such that an E occurs at a position if the number of 1's read in so far is even and an O if it is odd.	Input: 11100101 Output: OEOOOEEO

The system was implemented in Java (JDK 1.5.0_6) and all simulations were run on a Windows based 1.86 GHz PC with 1GB of RAM.

The random number generator used is that provided by the JDK library. The GP parameters used are tabulated in **Table 5.2**. Note that the application rates of the genetic operators were empirically derived. A different number of fitness cases were needed for each language and ranged from a minimum of 20 to a maximum of 56 fitness cases.

Table 5.2. GP Parameters

Population size	2000
Selection method	Tournament selection
Tournament size	5
Maximum number of nodes	6
Maximum generations	50
Crossover rate	85%
Mutation rate	5%
Reproduction rate	10%
Fitness cases	Pairs of input and corresponding output strings.
Raw fitness	The number of correct output strings.
Termination criteria	A solution has been found or 50 generations are completed.

The next section discusses the performance of the proposed GP system when applied to the benchmarks in **Table 5.1**.

6 Results and Discussion

The genetic programming system proposed in section **4** was applied to the data set in **Table 5.1**. The system was able to generate general solutions for all six machines. Ten runs were performed for each machine. Each solution was evolved in under a minute. The only other study applying evolutionary algorithms to the induction of finite transducers is that conducted by Lucas [8]. However, the domain for which transducers have been evolved is different from those presented in this paper and thus a direct comparison of the results is not possible. Hence, the evolved solutions are compared to human-generated solutions.

A solution for each of the machines and the corresponding "human generated" solutions are listed in **Table 6.1**. Note that for all six machines the evolved solutions are equivalent to the "human generated" solutions and for all of these machines the minimal transducer was evolved.

Table 6.2 lists the success rates for both standard and non-destructive operators.

The standard genetic operators implemented generally produced fit individuals and produced a 100% success rate for all machines accept M1. In the case of M1 the destructive effects of the genetic operators resulted in the lower success rate. The application of non-destructive operators for this machine produces a success rate of 100%.

Table 6.1. Mealy machine solutions

Machine	"Human Generated" Solution	GP Generated Solution
M1	b/0, a/0, q_0, q_1, a/0, q_2, b/0, a/1, b/0	b/0, a/0, 0, 1, a/0, 2, b/0, a/1, b/0
M2	b/0, a/0, a/0, q_0, q_1, a/0, q_2, b/0, b/1	b/0, a/0, a/0, 0, 1, a/0, 2, b/0, b/1
M3	0/0, 1/1, no carry, 0/1, start, 0/1, 1/0, carry, 1/0	0/0, 1/1, 1, 0/1, 0, 0/1, 1/0, 2, 1/0
M4	a/1, q_1, a/0, q_0, b/0, a/0, b/0, q_2, b/1	a/1, 1, a/0, 0, b/0, a/0, b/0, 2, b/1
M5	0/1, 1/0, q_0	0/1, 1/0, 0
M6	0/E, 1/O, 0/O, 1, 2, 1/E	0/E, 1/O, 0/O, 0, 1, 1/E

Table 6.2. Success Rates for Mealy Machine Simulations

Machine	Standard Operators	Non-destructive Operators
M1	70%	100%
M2	100%	100%
M3	100%	100%
M4	100%	100%
M5	100%	100%
M6	100%	100%

7 Conclusion

The main aim of the study presented in this paper is to assess the potential of genetic programming as a means of inducing finite transducers. A GP system, using directed graphs to represent transducers, was implemented and tested on six standard transducer problems. The results obtained are promising. The system was able to evolve solutions to all six problems. Furthermore, the solutions evolved were human competitive and in all cases the minimal transducer was found. Thus, the main contribution of this study is the discovery of a genetic programming system, using a simple direct representation of each transducer, as an effective methodology for generating finite transducers. Future extensions of this study will investigate applying the current system to specific applications in the domain of natural language processing.

Acknowledgments. The authors would like to thank the NRF (National Foundation for Research) of South Africa for funding this project.

References

1. Koza, J. R.: Genetic Programming I: On the Programming of Computers by Natural Selection, MIT Press (1992).
2. Fogel, L.J.: Owens, A., J., Walsh, M.J., Artificial Intelligence Through Simulated Evolution, Wiley and Sons, New York (1966).
3. Dupont, P.: Regular Grammatical Inference from Positive and Negative Samples by Genetic Search: the GIG Method. In Carrasco, R.C. and Oncina, J. (eds.): Grammatical Inference and Applications (ICGI-94). Springer-Berlin, Heidelberg (1994) 236 - 245.
4. Dunay, B.D.: Petry, F.E., Buckles, B.P: Regular Language Induction with Genetic Programming. In: Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA. IEEE Press (1994) 396 - 400.
5. Brave, S.: Evolving Deterministic Finite Automata Using Cellular Encoding. In : J.R. Koza et al. (eds.): Proceedings of the First Annual Conference on Genetic Programming (GP 96). MIT Press (1996) 39 - 44.

6. Luke, S., Hamahashi, S., Kitano, H.: "Genetic" Programming. In: Banzhaf, W., Daida, J., Eiben, A.E., Garzan, M. H., Honavar, Jakiela, V., M. and Smith, R.E.: Proceedings of the Genetic Programming and Evolutionary Computation Conference, Orlando, Florida, USA, Vol. 2. (1999) 1098 - 1105.
7. Lucas, S.M., Reynolds, T.: Learning DFA: Evolution versus Evidence Driven State Merging. In: The Proceedings of the 2003 Congress on Evolutionary Computation (CEC 2003). IEEE Press (2003) 351 – 358.
8. Lucas, S. M.: Evolving Finite State Transducers: Some Initial Explorations. In: Genetic Programming: 6th European Conference, EuroGP 2003, Essex, UK, April 14 -16, 2003, Lecture Notes in Computer Science, Vol. 2610. Springer (2003) 130 - 141.
9. Cohen, D. I. A.: Introduction to Computer Theory, John Wiley & Sons (1986).
10. Forcada, M.L.: Neural Networks: Automata and Formal Methods of Computation, January http://www.dlsi.ua.es/~mlf/nnafmc/pbook.pdf. (2002).

Author Index

Agapitos, Alexandros 291, 301
Andreae, Peter 55
Azad, R. Muhammad Atif 217

Banzhaf, Wolfgang 90
Bartoli, Alberto 170
Brabazon, Anthony 1
Budin, Leo 321
Buxton, Bernard 68

Chami, Malik 45
Ciesielski, Vic 281
Clergue, Manuel 45
Collard, Philippe 241

De Falco, Ivanoe 351
Decraene, James 361
Defoin Platel, Michael 45
Della Cioppa, Antonio 351
Di Chio, Cecilia 125, 331
Di Chio, Paolo 331
Dignum, Stephen 193
Downing, Richard M. 181

Edwards, Howard 23
Essam, Daryl 251

Fillon, Cyril 170
Flanagan, Colin 217
Folino, Gianluigi 160
Fonlupt, Cyril 12
Foster, James 33

Galván-López, Edgar 341
Gibbons, Adrian P. 148
Gilligan, Conor 1

Harding, Simon 90
Hauptman, Ami 78
Hemberg, Erik 1
Heywood, Malcolm I. 137, 229
Hoang, Tuan-Hao 251
Holladay, Kenneth 102

Jackson, David 148
Jakobović, Domagoj 321
Jelenković, Leonardo 321
Johnson, Colin G. 114

Kang, Moonyoung 251
Keijzer, Maarten 33
Kelly, Ciaran 361

Langdon, William B. 193
Lemczyk, Michal 229
Li, Xiang 281
Lichodzijewski, Peter 137
Lucas, Simon M. 291, 301

Mahler, Sébastien 12
Maisto, Domenico 351
Marion-Poty, Virginie 12
Mauri, Giancarlo 241
McKay, R.I. (Bob) 251
McMullin, Barry 361
Messom, Chris 23
Miller, Julian Francis 205, 261
Mitchell, George G. 361
Moraglio, Alberto 125
Mori, Naoki 251

Naidoo, Amashini 371
Nguyen, Xuan 251

O'Neill, Michael 1

Pillay, Nelishia 371
Pirola, Yuri 241
Pizzuti, Clara 160
Poli, Riccardo 125, 193

Raja, Adil 217
Robbins, Kay 102
Robilliard, Denis 12
Rodríguez-Vázquez, Katya 341
Ryan, Conor 217

Sarafopoulos, Anargyros 68
Scafuri, Umberto 351

Sekanina, Lukáš 311
Shin, Jungseok 251
Sipper, Moshe 78
Slaný, Karel 311
Smart, Will 55
Spezzano, Giandomenico 160

Tarantino, Ernesto 351
Tomassini, Marco 241

Vanneschi, Leonardo 241
Vérel, Sébastien 45, 241
von Ronne, Jeffery 102

Walker, James Alfred 205, 261
Walker, Matthew 23

Yamamoto, Lidia 271

Zhang, Mengjie 55