Michael O'Neill Leonardo Vanneschi
Steven Gustafson
Anna Isabel Esparcia Alcázar
Ivanoe De Falco Antonio Della Cioppa
Ernesto Tarantino (Eds.)

LNCS 4971

Genetic Programming

11th European Conference, EuroGP 2008
Naples, Italy, March 2008
Proceedings

Lecture Notes in Computer Science 4971

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Alfred Kobsa
University of California, Irvine, CA, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
University of Dortmund, Germany
Madhu Sudan
Massachusetts Institute of Technology, MA, USA
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Michael O'Neill Leonardo Vanneschi
Steven Gustafson
Anna Isabel Esparcia Alcázar
Ivanoe De Falco Antonio Della Cioppa
Ernesto Tarantino (Eds.)

Genetic Programming

11th European Conference, EuroGP 2008
Naples, Italy, March 26-28, 2008
Proceedings

Springer

Volume Editors

Michael O’Neill
University College Dublin, Ireland
E-mail: m.oneill@ucd.ie

Leonardo Vanneschi
Università degli Studi di Milano-Bicocca, Italy
E-mail: vanneschi@disco.unimib.it

Steven Gustafson
GE Global Research, Niskayuna, NY, USA
E-mail: steven.gustafson@research.ge.com

Anna Isabel Esparcia Alcázar
Instituto Tecnológico de Informática, Valencia, Spain
E-mail: aesparcia@iti.upv.es

Ivanoe De Falco
Ernesto Tarantino
ICAR–CNR, Naples, Italy
E-mail: {evostar, ernesto.tarantino}@na.icar.cnr.it

Antonio Della Cioppa
University of Salerno, Italy
E-mail: adellacioppa@unisa.it

Cover illustration: "Ammonite II" by Dennis H. Miller (2004-2005)
www.dennismiller.neu.edu

Library of Congress Control Number: 2008922954

CR Subject Classification (1998): D.1, F.1, F.2, I.5, I.2, J.3

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-78670-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-78670-2 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2008
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12242450 06/3180 5 4 3 2 1 0

Preface

The 11th European Conference on Genetic Programming, EuroGP 2008, took place in Naples, Italy from 26 to 28 March in the University of Naples Congress Centre with spectacular views over the Gulf of Naples. This volume contains the papers for the 21 oral presentations and 10 posters that were presented during this time. A diverse array of topics were covered reflecting the current state of research in the field of Genetic Programming, including the latest work on representations, theory, operators and analysis, evolvable hardware, agents and numerous applications.

A rigorous, double-blind peer review process was employed, with each submission reviewed by at least three members of the international Program Committee. In total 61 papers were submitted this year, making an acceptance rate of 34% for full papers, and an overall acceptance rate of 51% including posters. Submission of papers and the reviewing process were greatly assisted by the use of the MyReview management software originally developed by Philippe Rigaux, Bertrand Chardon and other colleagues from the Université Paris-Sud Orsay, France. We are especially grateful to Marc Schoenauer from INRIA, France for managing this system. Reviewers were asked to nominate keywords specifying their area of expertise, and these keywords were matched to those selected by the authors of the submitted papers with the assistance of the optimal assignment feature of the conference management software.

EuroGP 2008 was part of the larger Evo* 2008, which included three other co-located events, namely EvoCOP 2008, EvoBIO 2008, and EvoWorkshops 2008. We would like to take this opportunity to thank the many people who make EuroGP and Evo* a great success. Without the authors we would not have the high-quality submissions and presentations that make EuroGP such an interesting event. We extend our thanks to the Program Committee for their thorough, timely and constructive reviews that ensure the continued quality of EuroGP. We are indebted to the local organisers Antonio Della Cioppa, Ernesto Tarantino and Giuseppe Trautteur led by Ivanoe De Falco for their smooth organisation of the conference, in a spectacular location with many greatly-enjoyed social activities. We wish to wholeheartedly thank Professor Guido Trombetti, rector of the University of Naples "Federico II" and Professor Giuseppe Trautteur of the Department of Physical Sciences, who, with their extraordinary and invaluable support, made this event possible. Furthermore, we wish to express our most sincere gratitude to Naples City Council for supporting the local organisation and granting their patronage to the event. We also thank the Instituto Technológico de Informática, Valencia, Spain for hosting the Evo* website.

To our internationally renowned invited keynote speakers, Professor Emeritus Hans-Paul Schwefel (Dortmund University of Technology, Germany), and Dr.

Stefano Nolfi (Institute of Cognitive Science and Technologies, CNR, Italy), we express our sincere gratitude.

Last and certainly not least, we especially thank Jennifer Willies and the Centre for Emergent Computing at Napier University. Without their continued dedication and coordination, this event would not be possible.

March 2008

Michael O'Neill
Leonardo Vanneschi
Steven Gustafson
Anna I. Esparcia Alcázar
Ivanoe De Falco
Antonio Della Cioppa
Ernesto Tarantino

Organization

Administrative details were handled by Jennifer Willies, Centre for Emergent Computing at Napier University, Scotland, UK.

Organizing Committee

Program Co-chairs	Michael O'Neill (University College Dublin, Ireland) Leonardo Vanneschi (Università degli Studi di Milano-Bicocca, Italy)
Publication Chair	Steven Gustafson (GE Global Research, USA)
Publicity Chair	Anna I. Esparcia Alcázar (Instituto Tecnológico de Informática, Spain)
Local Co-chairs	Ivanoe De Falco (ICAR-CNR, Italy) Antonio Della Cioppa (University of Salerno, Italy) Ernesto Tarantino (ICAR-CNR, Italy)

Program Committee

Hussein Abbass, UNSW@ADFA, Australia
Lee Altenberg, University of Hawaii at Manoa, USA
R. Muhammad Atif Azad, University of Limerick, Ireland
Wolfgang Banzhaf, Memorial University of Newfoundland, Canada
Anthony Brabazon, University College Dublin, Ireland
Nicolas Bredeche, Université Paris-Sud, France
Edmund Burke, University of Nottingham, UK
Stefano Cagnoni, Università degli Studi di Parma, Italy
Antonio Della Cioppa, University of Salerno, Italy
Philippe Collard, Laboratoire I3S (UNSA-CNRS), France
Pierre Collet, LSIIT-FDBT, France
Ernesto Costa, Universidade de Coimbra, Portugal
Michael Defoin Platel, University of Auckland, New Zealand
Edwin DeJong, Universiteit Utrecht, The Netherlands
Ian Dempsey, Pipeline Financial Group, Inc., USA
Federico Divina, Universidad Pablo de Olavide, Spain
Marc Ebner, Universität Würzburg, Germany
Anikó Ekárt, Aston University, UK
Anna Esparcia-Alcázar, ITI Valencia, Spain
Daryl Essam, UNSW@ADFA, Australia
Francisco Fernández de Vega, Universidad de Extremadura, Spain
Christian Gagné, MDA, Canada
Mario Giacobini, Università degli Studi di Torino, Italy
Folino Gianluigi, ICAR-CNR, Italy
Steven Gustafson, GE Global Research, USA

Jin-Kao Hao, LERIA, Université d'Angers, France
Inman Harvey, University of Sussex, UK
Tuan-Hao Hoang, University of New South Wales @ ADFA, Canada
Gregory Hornby, UCSC, USA
Colin Johnson, University of Kent, UK
Tatiana Kalganova, Brunel University, UK
Maarten Keijzer, Chordiant Software International, The Netherlands
Robert E. Keller, University of Essex, UK
Graham Kendall, University of Nottingham, UK
Asifullah Khan, Pakistan Inst. of Engineering and Applied Sciences, Pakistan
Krzysztof Krawiec, Poznan University of Technology, Poland
Jiri Kubalik, Czech Technical University in Prague, Czech Republic
William B. Langdon, University of Essex, UK
Kwong Sak Leung, The Chinese University of Hong Kong, Hong Kong
John Levine, University of Strathclyde, UK
Simon M. Lucas, University of Essex, UK
Robert Matthew MacCallum, Imperial College London, UK
Penousal Machado, Universidade de Coimbra, Portugal
Bob McKay, Seoul National University, Korea
Nic McPhee, University of Minnesota, Morris, USA
Jörn Mehnen, Cranfield University, UK
Xuan Hoai Nguyen, Seoul National University, Korea
Miguel Nicolau, INRIA, France
Julio Cesar Nievola, Pontificia Universidade Catolica do Parana, Brazil
Michael O'Neill, University College Dublin, Ireland
Una-May O'Reilly, MIT, USA
Clara Pizzuti, Institute for High Performance Computing and Networking, Italy
Riccardo Poli, University of Essex, UK
Thomas Ray, University of Oklahoma, USA
Denis Robilliard, Université du Littoral, Cote D'Opale, France
Marc Schoenauer, INRIA, France
Michele Sebag, Université Paris-Sud, France
Lukas Sekanina, Brno University of Technology, Czech Republic
Yin Shan, Medicare, Australia
Moshe Sipper, Ben-Gurion University, Israel
Alexei N. Skurikhin, Los Alamos National Laboratory, USA
Terence Soule, University of Idaho, USA
Ivan Tanev, Doshisha University, Japan
Ernesto Tarantino, ICAR-CNR, Italy
Marco Tomassini, University of Lausanne, Switzerland
Leonardo Vanneschi, Università degli Studi di Milano, Italy
Sébastien Verel, Université de Nice Sophia Antipolis/CNRS, France
Man Leung Wong, Lingnan University, Hong Kong
Tina Yu, Memorial University of Newfoundland, Canada
Mengjie Zhang, Victoria University of Wellington, New Zealand

Table of Contents

Oral Presentations

Training Time and Team Composition Robustness in Evolved Multi-agent Systems 1
Russell Thomason, Robert B. Heckendorn, and Terence Soule

Winning Ant Wars: Evolving a Human-Competitive Game Strategy Using Fitnessless Selection 13
Wojciech Jaśkowski, Krzysztof Krawiec, and Bartosz Wieloch

In Silicon No One Can Hear You Scream: Evolving Fighting Creatures 25
Thomas Miconi

Real-Time, Non-intrusive Speech Quality Estimation: A Signal-Based Model 37
Adil Raja and Colin Flanagan

Good News: Using News Feeds with Genetic Programming to Predict Stock Prices 49
Fiacc Larkin and Conor Ryan

A Genetic Programming Approach to Deriving the Spectral Sensitivity of an Optical System 61
Marc Ebner

A SIMD Interpreter for Genetic Programming on GPU Graphics Cards 73
W.B. Langdon and Wolfgang Banzhaf

Partitioned Incremental Evolution of Hardware Using Genetic Programming 86
David Jackson

Population Parallel GP on the G80 GPU 98
Denis Robilliard, Virginie Marion-Poty, and Cyril Fonlupt

Operator Equalisation and Bloat Free GP 110
Stephen Dignum and Riccardo Poli

Practical Model of Genetic Programming's Performance on Rational Symbolic Regression Problems 122
Mario Graff and Riccardo Poli

Semantic Building Blocks in Genetic Programming 134
Nicholas Freitag McPhee, Brian Ohs, and Tyler Hutchison

A Simple Powerful Constraint for Genetic Programming 146
Gearoid Murphy and Conor Ryan

Crossover, Sampling, Bloat and the Harmful Effects of Size Limits 158
Stephen Dignum and Riccardo Poli

The Performance of a Selection Architecture for Genetic Programming 170
David Jackson

A Comparison of Cartesian Genetic Programming and Linear Genetic Programming 182
Garnett Wilson and Wolfgang Banzhaf

Evolvability Via Modularity-Induced Mutational Focussing 194
Richard M. Downing

A Linear Estimation-of-Distribution GP System 206
Riccardo Poli and Nicholas Freitag McPhee

Feature Discovery in Reinforcement Learning Using Genetic Programming 218
Sertan Girgin and Philippe Preux

Hardware Accelerators for Cartesian Genetic Programming 230
Zdenek Vasicek and Lukas Sekanina

Genetic Programming and Class-Wise Orthogonal Transformation for Dimension Reduction in Classification Problems 242
Kourosh Neshatian and Mengjie Zhang

Posters

Evolving Proactive Aggregation Protocols 254
Thomas Weise, Michael Zapf, and Kurt Geihs

GP Classification under Imbalanced Data Sets: Active Sub-sampling and AUC Approximation 266
John Doucette and Malcolm I. Heywood

Exposing a Bias Toward Short-Length Numbers in Grammatical Evolution 278
Marco A. Montes de Oca

Cooperative Problem Decomposition in Pareto Competitive Classifier Models of Coevolution 289
Andrew R. McIntyre and Malcolm I. Heywood

Integrating Categorical Variables with Multiobjective Genetic Programming for Classifier Construction 301
Khaled Badran and Peter Rockett

The Effects of Constant Neutrality on Performance and Problem Hardness in GP 312
Edgar Galván-López, Stephen Dignum, and Riccardo Poli

Applying Cost-Sensitive Multiobjective Genetic Programming to Feature Extraction for Spam E-mail Filtering 325
Yang Zhang, HongYu Li, Mahesan Niranjan, and Peter Rockett

PlasmidPL: A Plasmid-Inspired Language for Genetic Programming 337
Lidia Yamamoto

Using Genetic Programming for Turing Machine Induction 350
Amashini Naidoo and Nelishia Pillay

Altering Search Rates of the Meta and Solution Grammars in the mGGA 362
Erik Hemberg, Michael O'Neill, and Anthony Brabazon

Author Index 375

Training Time and Team Composition Robustness in Evolved Multi-agent Systems

Russell Thomason, Robert B. Heckendorn, and Terence Soule

University of Idaho, Department of Computer Science,Moscow, ID 83844-1010
thom0398@uidaho.edu, heckendo@uidaho.edu, tsoule@cs.uidaho.edu

Abstract. Evolutionary algorithms are effective at creating cooperative, multi-agent systems. However, current Island and Team algorithms show subtle but significant weaknesses when it comes to balancing member performance with member cooperation, leading to sub-optimal overall team performance. In this paper we apply a new class of cooperative multi-agent evolutionary algorithms called Orthogonal Evolution of Teams (OET) which produce higher levels of cooperation and specialization than current team algorithms. We also show that sophisticated behavior evolves much sooner using OET algorithms, even when training resources are significantly reduced. Finally, we show that when teams must be reformed, due to agent break down for example, those teams composed of individuals sampled from OET teams perform much better than teams composed of individuals sampled from teams created by other methods.

1 Introduction

Many problems require solutions using teams of multiple agents working together to achieve a goal, such as robot soccer or Serengeti world [8,9]. Other problems like autonomous robot exploration use teams to examine the search space more effectively. However, producing efficient and effective teams is difficult. Teams may need tens, thousands, or in the case of nanotechnology, millions of members performing simple tasks. Or teams may be small, but composed of agents capable of performing many tasks. In the case of heterogeneous teams, the members might have different abilities and sub-goals. Thus, finding algorithms that automate the process of training is vitally important and evolutionary algorithms that can evolve successful teams incorporating specialization and cooperation simultaneously would represent an important advance.

We are also interested in evolving specific forms of robustness. 1) Robustness with regard to team size (team size scaling), i.e. the ability of teams to effectively scale upwards in size as more agents are needed. 2) Robustness with regard to available resources (in this case, time scaling), i.e. the ability to quickly evolve sophisticated behavior when training resources are significantly limited. 3) Robustness with regards to functionality, i.e. the ability to perform and cooperate well with members that were not trained together, for example if teams had to be reformed due to agent break down.

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 1–12, 2008.
© Springer-Verlag Berlin Heidelberg 2008

Our new class of cooperative, multi-agent evolutionary algorithms (OET) are specifically designed to address these issues (specialization, cooperation and robustness) by applying evolutionary pressure on both individual members, which leads to specialization, and evolutionary pressure on teams, which leads to cooperation. In this paper we show that using both types of pressure produces teams that meet our robustness goals as well.

2 Background

Evolutionary algorithms are often used to evolve teams of agents, where the goal is to maximize utility or solve a task, what Panait and Luke call cooperative multi-agent learning [6]. Even homogeneous teams, team whose members have identical capabilities, often benefit from control structures that allow agents to specialize in different subdomains of the problem [5,15]. Heterogeneous teams, teams whose members have different capabilities, *require* control structures that allow agents with different abilities to operate in the same team while also ensuring the team itself does well. Programming the behavior between multiple agents with different abilities to optimize cooperation is extremely difficult. For these reasons, evolutionary algorithms are often used to train teams, and the agents within a team are usually evolved together to increase cooperation.

Evolutionary approaches for training multi-agent teams have been successfully applied to a wide range of knowledge representations, including teams of: neural networks[7], oblique decision trees[1], and stack-based predictors[11]. Evolutionary approaches have also been applied to a wide range of problem domains including robot navigation [4], team sporting strategies [12], predator strategies [3,9], hazard assessment [10], and cancer and diabetes diagnosis [1,7].

Cooperative evolutionary algorithms generally fit into two groups, Island or Team algorithms. Research suggests Island approaches produce teams of strong individuals that cooperate poorly, and Team approaches produce teams of weak individuals that cooperate strongly [5,15]. Ideally, teams should be composed of strong individuals that cooperate well. In order to overcome the weaknesses of Island and Team approaches, Soule introduced a new class of cooperative multi-agent evolutionary algorithms called Orthogonal Evolution of Teams (OET) that alternately apply pressure on teams and individuals during selection and replacement [5,15]. Soule described this class of algorithms as orthogonal because they alternate between two orthogonal views of the population: as a single population of teams of size N and as a set of N independent populations of individuals. Thomason described two main variations of OET and compared their performance to Team and Island algorithms using classification problems [15]. It was shown that OET produces teams whose members perform better than those generated with Team approaches and which cooperate better than those generated using Island approaches. Increased individual performance and team cooperation led to significantly better team performance. In addition, Soule and Heckendorn found that OET teams are significantly more robust with regards to team size than teams evolved using the Island and Team approaches [13,14].

In this paper we extend the research to determine which of the three general approaches (Team and two OET varients) produce teams that are the most robust with regard to resources (specifically time) and with regard to cooperation with members from other teams. To test limited resource robustness we significantly reduced the amount of time the teams were allowed to train, as compared to how long they would operate in the environment during testing. As a novel test of member robustness we measure how well teams perform when their members are randomly replaced by members from other teams. Our results show that OET performs significantly better than Team algorithms on both of these tests.

3 The Problem Environment

The environment is a two dimensional grid composed of 2025 (45x45) squares. At the beginning of each evaluation exactly twenty percent of the squares are labeled as interesting. The interesting squares are determined randomly for each evaluation so that agents cannot memorize where the interesting squares are, instead the agents must learn general search algorithms.

There are two agent types: scouts and investigators. The scouts role is to find interesting squares and mark them with a beacon that is detectable at a distance by the investigators. The investigators role is to investigate interesting squares. Scouts travel at up to twice the speed of investigators. If a scout is in or next to an interesting square, it automatically places a beacon in the interesting square (unless there is already a beacon there). If an investigator enters an interesting square, regardless of whether the square is marked with a beacon, it changes the square to investigated and deactivate the beacon, if any, in the square. Neither type of agent can sense interesting squares at a distance without a beacon. Thus, the teams must evolve general search behaviors.

Since the agents can see the beacons (if any have been placed) at a distance, the space can be more efficiently explored by the fast moving scouts marking interesting areas and the slower investigators using the beacons to go directly to the areas to be investigated. Thus, the two types of agents have different subgoals and they must divide up the space to be searched efficiently since the task has a time limit. If they all search in the same area they will fail to search the entire space.

This model represents an abstraction of a number of practical problems. For example, scouts and investigators could represent two robot types exploring a minefield. Scouts fly overhead marking locations of potential mines and investigators deactivate the mines. Alternatively, they could represent an automated planetary surveying team. Scouts identify potentially interesting geological formations and investigators follow up by taking soil samples, etc.

The agent environment is a two-dimensional, real-valued space, so agent's have real vlaued location within a square. Agent movement is determined by an expression tree that returns a vector, which represents the direction and speed the agent will travel. However, investigators are limited to moves of length one and scouts are limited to moves of length two.

Input vectors (terminal nodes in the expression tree):

1. North - A unit vector pointing North ($\pi/2$ radians).
2. Constant - A vector that is generated randomly (magnitude in [0,2], direction in [0,2π radians]) when the node is created. It remains during the lifetime of the agent, but it can change through mutation.
3. Random - A vector that is randomized each time it is evaluated (magnitude in [0,2], direction in [0,2π radians]).
4. Nearest Scout - A vector from the agent to nearest scout.
5. Nearest Investigator - A vector from the agent to nearest investigator.
6. Nearest Beacon - A vector from the agent to nearest beacon.
7. Nearest Edge - A vector from the agent to search space nearest boundary.
8. Last Move - A vector representing the agent's last move.
9. Check Bounds - A zero magnitude vector with a small, arbitrary, positive direction if inside search space and a small, arbitrary, negative direction otherwise.

In this implementation there is no limit for detecting a beacon. If an input is meaningless, e.g. nearest beacon when no beacons are present, then a random vector is returned. The nearest edge also accounts for the possibility that the agent is outside the search space, although agents must figure out for themselves how to remain within bounds.

Vector operations (non-terminal nodes in the expression tree):

1. Add - Takes 2 vector arguments and returns the vector sum.
2. Invert - Takes 1 vector argument and returns a vector with its direction inverted (by adding π radians).
3. If-Less-Than-Else-Magnitude - Takes four vector arguments. If the magnitude of vector 1 is less than the magnitude of vector 2, then return vector 3, otherwise return vector 4.
4. If-Less-Than-Else-Direction - Takes four vector arguments. If the angle of vector 1 is less than the angle of vector 2, then return vector 3, otherwise return vector 4.

3.1 Fitness Evaluation

Each iteration during evolution consists of a simulation with a fixed number of time steps. All of the agents start at random, real valued, locations within the center square. Each agent has its input vectors updated and then its expression tree is evaluated so the agent can move. The agents move sequentially and update their input vectors whenever an agent moves.

Scouts gain a point for placing a beacon in an interesting square (only one beacon per square is allowed), and investigators gain a point by investigating an interesting square (even if no beacon is present). Beacons are removed after a square is investigated, and it cannot be re-flagged. Agents are penalized 0.1 points for each time step they remain outside of the search space. Because the environment changes for each simulation the fitness of an agent will vary

somewhat between evaluations. The fitness of the team is the sum of the agent fitnesses. There are 2025 squares in the environment and 405 interesting squares per simulation. Thus, the maximum team score is 810 points if the scouts cover all the interesting squares with beacons, and the investigators investigate them all, and no agent ever leaves the search space (which is very unlikely).

4 Evolutionary Algorithms

A steady state population model is used. The parameters are listed in Table 1. In the equal time experiments the teams are trained and tested using the same number of time steps (the amount of time they spend in the environment each iteration). During the time scaling experiments, the teams are trained with much less time in the environment then they have when they are tested. Every iteration consists of $PopulationSize/2$ rounds of parent selection and replacement.

Table 1. Summary of the evolutionary algorithm parameters

Population Size	100
Team Size	3 scouts and 3 investigators
Selection and Replacement	3 member tournament
Mutation Probability	2 / tree size
Crossover Probability	1.0
Iterations	250
Trials	40
Training/Testing Time Steps	200/200, 300/300, 400/400 or 200/400, 300/400

In the Team algorithm all evolutionary pressure occurs at the level of teams. During selection two teams are chosen through a tournament to be parents. The parents are crossed over to produce two offspring which are then mutated. A reverse tournament is performed to find two poor teams for replacement. Offspring replace the poor teams if their team fitness is higher. Therefore, teams are selected and replaced based on their team fitness. Team level pressure leads to cooperation, but there is no direct pressure to increase the fitness of individual team members.

We have not included data from an Island algorithm because team problems, especially ones which rely on cooperation, require team algorithms. A hybrid island algorithm was tested in the previous two papers [13,14], and it did not perform well. If members are evolved in independent populations, it is unlikely they will produce any sophisticated cooperative behaviors when combined for testing, because the team members will never have trained together.

In OET1, selection is done on individuals and replacement is done on teams. Offspring are created by making an empty team and adding fit individuals one at a time by treating the single population of N-sized teams as N independent populations of individuals and doing tournament selection within each population. Therefore, the first team member is chosen from the population that

represents all of the first members from each team; the second team member is chosen from the population that represents all of the second members from each team, and this continues until the new team is filled. Two teams are constructed and undergo crossover and mutation. The offspring are evaluated as teams and replacement is done by comparing team fitness. Team members must have high fitness to be selected for a parent team, and a team in the population must have a high fitness to avoid being selected for replacement.

In OET2, selection is done on teams and replacement is done on individuals. Two fit teams are selected to be parents by tournament selection. Their members undergo crossover and mutation to produce two new offspring teams. Replacement is done by comparing the fitness of individuals in the offspring to the fitness of individuals in the population. This is done by treating the population of N-sized teams as N independent populations of individuals. Poor individuals are selected for replacement by individuals in the new offspring. A team must have high fitness to be selected as a parent and team members must have a high fitness to avoid being selected for replacement.

Therefore, the OET algorithms apply direct pressure to individuals and teams through selection and replacement. However, special consideration must be given to OET2 because it replaces individuals during the replacement phase, which means team fitness can become inaccurate. Potentially, the entire population would need to be re-evaluated to update all the team fitnesses. Although this is one option, it is undesirable because it increases the amount of evaluations needed as compared to the OET1 and Team algorithms, so we decided to simply not update team fitness at every iteration for OET2. The team fitnesses will become out of date, but all the teams will be equally out of date, and because most of the members have not been replaced the team fitness is reasonably accurate. Then, every 25 iterations we skip a parent selection and replacement phase and just update the entire population. This results in no increase of evaluations needed because during a normal iteration the number of evaluations used is equal to the population size because that many offspring are made. It also means that only 4% of the iterations are used to update team fitnesses in the population.

5 Results

All results are the average of 40 independent trials. The results are presented in three sections. First, are the results of the equal time experiments where training and testing time steps are the same. Second, are the results of the time scaling experiments where the training time steps are much smaller than the testing time steps. Finally, are the results from a new type of cooperation and robustness test which measures how well individuals perform in their own team versus a random team. We also computed the P-values for the two tailed Student's t-test for each pair of algorithms for each combination of training and testing time steps. All values are significant (below 0.01) except the difference between OET1 and OET2 on the 200/200 test, which was only below 0.05. This confirms that the differences between the average performance presented in the following sections are in fact significant.

5.1 Equal Time Experiments

In this experiment training and testing time steps are equal. Table 2 shows the average team fitness and standard deviation. The difference in performance between the Team algorithm and the OET algorithms are especially noticeable. The OET algorithms perform very similarly and both outperform the Team algorithm by an average of 20% in the 200/200 test, 16% in the 300/300 test, and 10% in the 400/400 test.

The OET algorithms do especially well with limited time resources because pressure is being applied to individuals, which forces them to spread out earlier so they can gain points by dropping beacons or investigating interesting squares. Additionally, the pressure applied to teams produces interesting cooperative behaviors. Often the OET algorithms would produce teams where the agents would break out into sub-teams of investigator/scout pairs, where the fast moving scouts would move in circular patterns around the investigators. This allowed the scouts to drop many beacons that would be relatively close to its nearby investigator. We suspect that this type of behavior, e.g. spreading out early, but doing so in a way that promotes cooperation, is only consistently reproducible through algorithms that apply direct evolutionary pressure on individuals and teams.

Table 2. Average team fitness for the equal time experiments

Algorithm	Training TS	Testing TS	Avg Team Fitness
OET1	200	200	585.9 (20.9)
OET2	200	200	577.7 (13.7)
TEAM	200	200	483.6 (54.8)
OET1	300	300	718.3 (16.2)
OET2	300	300	708.7 (13.8)
TEAM	300	300	617.6 (59.6)
OET1	400	400	776.7 (8.8)
OET2	400	400	762.0 (11.2)
TEAM	400	400	701.8 (59.4)

Table 3 shows, for both the scouts and investigators, the average worst, averages, and average best fitnesses. The average worst investigator and scout in the teams produced by OET algorithms are almost twice as fit as those evolved from the Team algorithm. The average best investigator and scout usually comes from the Team algorithm which shows that most of these teams have one very fit scout and investigator while the rest of the team members are mostly riding its coattails. The OET algorithms produce investigators which are on average 22% more fit in the 200/200 tests, 18% more fit in the 300/300 tests, and 11% more fit in the 400/400 tests than what the Team algorithm produces. The OET algorithms produce scouts which are on average 19% more fit in the 200/200 tests, 13% more fit in the 300/300 tests, and 8% more fit in the 400/400 tests than the what the Team algorithm produces.

Table 3. Average scout and investigator fitness for the equal time experiments showing the 200/200, 300/300 and 400/400 tests

	Investigators			**Scouts**		
Algorithm	**Avg Min**	**Avg**	**Avg Max**	**Avg Min**	**Avg**	**Avg Max**
OET1	81.8	90.2	98.1	85.9	105.1	126.9
OET2	75.3	84.6	93.3	89.3	107.9	127.5
TEAM	43.3	71.4	93.0	50.7	89.8	129.1
OET1	107.6	119.5	132.1	88.1	119.9	149.9
OET2	103.4	117.0	130.0	97.5	119.2	143.1
TEAM	60.5	100.5	130.2	58.5	105.4	154.4
OET1	113.1	131.3	148.7	91.3	127.6	161.1
OET2	113.0	128.9	145.0	95.7	125.0	153.1
TEAM	59.6	117.3	161.8	58.1	116.7	189.3

We believe that the bigger difference in investigator performance is due to the way that scouts cooperate with investigators in the OET algorithms. The paired behavior described above, in which a scout circles an investigator, means the investigators have more opportunities to investigate interesting squares since they are next to more beacons that are placed by their nearby scout. Overall, the data from Table 2 shows that the higher average individual fitness of the OET algorithm is leading to higher average team fitness and that it is a function of the sophisticated cooperation behaviors we observed.

5.2 Time Scaling Experiments

In these experiments training time is shorter than testing time. This addresses two questions. First, do the teams with more time in the first set of experiments perform well because they were operating in the environment longer or because they evolved better search abilities? Second, can training efficiency be improved by training for short periods of time? If the difference in fitness averages between the time scaling tests and the equal time tests were large, this would imply that some behaviors need more time to evolve. Alternatively, if the teams evolved under shorter training periods perform equally well during longer tests it shows that evolutionis robust with respect to trainging time.

Table 4 shows the average team fitness and standard deviation in the time scaling experiments. Again, the OET algorithms perform very well and produce teams that are on average 15% more fit in the 200/400 tests and about 17% more fit in the 300/400 tests than what the Team algorithm produces. Clearly, the Team algorithm needs as much training time as possible, as their results improve greatly with time. In contrast, the OET algorithms perform almost as well as the 400/400 tests with significantly fewer training time steps. This means that OET algorithms produce teams that are fairly robust with respect to time resources, and thus high fitness teams can be produced much quicker with the OET algorithms. It also shows that their cooperative behavior and general search techniques evolve relatively quickly.

Table 4. Average team fitness for the time scaling experiments (400/400 tests included for comparison)

Algorithm	Training Time	Testing Time	Avg Team Fitness
OET1	200	400	747.8 (15.0)
OET2	200	400	764.0 (11.1)
TEAM	200	400	658.7 (56.0)
OET1	300	400	746.8 (19.4)
OET2	300	400	765.8 (10.9)
TEAM	300	400	646.3 (59.0)
OET1	400	400	776.7 (8.8)
OET2	400	400	762.0 (11.2)
TEAM	400	400	701.8 (59.4)

The minimum, maximum, and average performance of individual scouts and investigators in the time scaling experiments (data not shown) produced the same trends as in Table 3. The average best investigator and scout usually comes from the Team algorithm, while the OET algorithms produce investigators and scouts with higher average fitness.

5.3 Cooperation Tests

Finally, we performed two cooperation tests. During each of the previous experiments the best team from each trial was saved, which resulted in a pool of 40 teams from each experiment. Random teams were formed using two different methods and tested. The goal was to measure cooperation by testing how well individuals perform in random teams versus the team they evolved in. In addition, this tests robustness with regards to team composition.

In the first cooperation test the 40 teams were pooled with all investigators in one set and all scouts in another set. 120 random teams were formed by selecting three investigators (with replacement) and three scouts (with replacement). In the second cooperation test we used the same 40 teams, but kept the investigators and scouts in their respective columns. So all of the first investigators from each team formed a list, all of the second investigators from each team formed a list, and this continued until there were 6 lists (3 for investigators and 3 for scouts). Then we formed another 120 random teams by selecting (with replacement) an agent from each list.

Forming random teams using both methods allows us to determine whether team members in particular positions within a team consistently develop similar roles. In all three algorithms individuals maintain their position within a team during evolution, e.g. the first scout in a team is always the first scout and is always crossed with other first scouts. This makes it easier for members to evolve specialized roles [2]. For example, a simple form of specialization might simply be that the first scout always begins by exploring in the Northeast direction, and this behavior will eventually become fixed in all members of a population. By using the two methods described above we can determine whether similar specialized

Table 5. Results of cooperation tests showing the 200/200, 300/300, and 400/400 tests. Average of 120 randomly sampled teams. The percent drop is compared to team fitness in Table 2.

		Test 1		**Test 2**	
timesteps	**Algorithm**	**Avg Team**	**%Drop**	**Avg Team**	**%Drop**
200/200	OET1	436.6 (67.9)	25	439.8 (74.7)	25
	OET2	492.8 (59.8)	15	491.2 (54.6)	15
	TEAM	320.1 (97.9)	34	319.6 (86.1)	34
300/300	OET1	576.8 (76.1)	20	585.4 (62.7)	19
	OET2	628.5 (44.0)	11	629.0 (42.3)	11
	TEAM	409.6 (124.6)	34	400.2 (144.9)	35
400/400	OET1	660.3 (95.7)	15	643.4 (99.5)	17
	OET2	692.1 (89.2)	9	699.3 (48.1)	8
	TEAM	476.9 (172.0)	32	465.8 (176.3)	34

behaviors evolve between trials. If they do then preserving members positions by having six pools (method 2 above) should yield better results. Otherwise, if specialized roles evolve completely independently across multiple trials then the two methods should produce similar results.

Table 5 shows the results of both cooperation tests. Both OET algorithms did significantly better when random teams were formed, and OET2 was noticeably better than OET1. For cooperation test 1, random teams from the Team algorithm drop in fitness by an average of 33.8%, while random teams from the OET1 algorithm drop in fitness by an average of 20.2%, and random teams from the OET2 algorithm only drop in fitness by an average of 11.5%. That the most significant drop is with the Team algorithm is reasonable, because the composition of teams in the Team algorithm remains constant during evolution.

In contrast, in OET1, new teams are created by combining copies of the best members in the population. These new teams will only be successful if the members cooperate well. Thus, there is pressure not only to evolve members that perform well and that cooperate within their team, but also to evolve members that perform well when combined with novel members. In OET2, replacement inserts individuals into different teams within the population. Therefore, individuals are very mobile within the population and teams must evolve to successfully accommodate the insertion of new individuals to be successful. In general, both OET algorithms evolve extra-team cooperation naturally because OET1, and especially OET2, places a higher order pressure on the population to maintain individuals that cooperate well inside their own team, but also cooperate well with similar individuals from other teams.

Finally, the results with the two cooperation tests are essentially the same. This confirms that the specialized roles evolved by team members in a particular position are independent between trials. That is, the evolved role of a scout in position 1 in one trial (say searching the Northeast corner or circling the investigator in position 3) is completely independent of the roles it evolves in other trials, which is what we expected.

Table 6. Results of cooperation tests showing the 200/400, 300/400, and 400/400 tests. Average of 120 randomly sampled teams. The percent drop is compared to team fitness in Table 4.

		Test 1		Test 2	
Timesteps	**Algorithm**	**Avg Team**	**%Drop**	**Avg Team**	**%Drop**
200/400	OET1	647.9 (102.5)	13	643.0 (86.7)	14
	OET2	683.6 (63.0)	11	681.4 (82.4)	11
	TEAM	470.7 (165.6)	29	492.0 (162.6)	25
300/400	OET1	628.1 (121.2)	16	631.8 (128.4)	15
	OET2	695.1 (64.1)	9	687.6 (71.4)	10
	TEAM	431.3 (190.0)	33	453.7 (170.5)	30
400/400	OET1	660.3 (95.7)	15	643.4 (99.5)	17
	OET2	692.1 (89.2)	9	699.3 (48.1)	8
	TEAM	476.9 (172.0)	32	465.8 (176.3)	34

Table 6 shows the results of the cooperation tests from the time scaling experiments. Randomly sampled teams from the Team algorithm drop in fitness by an average of 30.5%, while randomly sampled teams from the OET1 algorithm drop in fitness by an average of 15.0%, while randomly sampled teams from the OET2 algorithm only drop in fitness by an average of 9.7%.

6 Conclusion

Our results lead to three important conclusions. First, the equal time experiments showed that OET algorithms significantly outperform standard Team approaches on a complex multi-agent problem. Second, the time scaling experiments showed that OET algorithms are very robust to limited training resources. The teams produced by OET performed almost as well when their training time was cut in half, while the Team approach needed as much training time as possible. This is significant because in many real world applications an agent's time in the field is likely to be substantially longer than the time available for training. Third, the cooperation tests showed that OET algorithms, especially OET2, significantly outperforms other approaches when teams must be reformed and that a significant amount of cooperation is maintained between all individuals in the population. This team member robustness is important because it allows teams to be successfully reformed if some members fail or are damaged.

We also observed sophisticated behavior from OET teams that evolved relatively quickly. Members learned to spread out to avoid covering the same areas and formed investigator/scout pairs where the fast moving scouts moved in circular patterns around the investigators. This cooperation increased the average fitness of scouts, and especially, investigators. This shows that OET algorithms are not just outperforming the team algorithms, but are doing so by evolving effective cooperative behaviors. The speed with which these behaviors evolve is also very promising as it strongly suggests that even with increasingly complex agents and environments, evolutionary algorithms such as OET will be able to generate effective cooperative behaviors in a reasonable amount of time.

References

1. Cantu-Paz, E., Kamath, C.: Inducing oblique decision trees with evolutionary algorithms. IEEE Transactions on Evolutionary Computation 7(1), 54–68 (2003)
2. Haynes, T., Sen, S.: Evolving behavioral strategies in predators and prey. In: Weiß, G., Sen, S. (eds.) Adaptation and Learning in Multiagent Systems. LNAI, Berlin, Germany, Springer, Heidelberg (1995)
3. Haynes, T., Sen, S., Schoenefeld, D., Wainwright, R.: Evolving a team. In: Siegel, E.V., Koza, J.R. (eds.) Working Notes for the AAAI Symposium on Genetic Programming. AAAI, 10–12 November, pp. 23–30. MIT, Cambridge (1995)
4. Iba, H.: Multiple-agent learning for a robot navigation task by genetic programming. In: Koza, J.R., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M., Iba, H., Riolo, R.R. (eds.) Genetic Programming 1997: Proceedings of the Second Annual Conference, pp. 195–200. Morgan Kaufmann, San Francisco (1997)
5. Komireddy, P., Soule, T. (eds.): Orthogonal Evolution of Teams: A Class of Algorithms for Evolving Teams with Inversely Correlated Errors (2006)
6. Luke, S., Panait, L.: Cooperative multi-agent learning: The state of the art. Autonomous Agents and Multi-agent Systems 11(3), 387–434 (2005)
7. Liu, Y., Yao, X., Higuchi, T.: Evolutionary ensembles with negative correlation learning. IEEE Transactions on Evolutionary Computation 4(4), 380–387 (2000)
8. Luke, S., Hohn, C., Farris, J., Jackson, G., Hendler, J.: Co-evolving soccer softbot team coordination with genetic programming. In: Proceedings of the First International Workshop on RoboCup, at the International Joint Conference on Artificial Intelligence, Nagoya, Japan (1997)
9. Luke, S., Spector, L.: Evolving teamwork and coordination with genetic programming. 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, 28–31 July, pp. 150–156. MIT Press, Cambridge (1996)
10. Obitz, D.W., Basak, S.C., Gute, B.D.: Hazard assessment modeling: An evolutionary ensemble approach. In: Proceedings of the Genetic and Evolutionary Computation Conference: GECCO-1999, pp. 1543–1650. Morgan Kaufmann, San Francisco (1999)
11. Platel, M.D., Chami, M., Clergue, M., Collard, P.: Teams of genetic predictors for inverse problem solving. In: Keijzer, M., Tettamanzi, A.G.B., Collet, P., van Hemert, J.I., Tomassini, M. (eds.) EuroGP 2005. LNCS, vol. 3447, Springer, Heidelberg (2005)
12. Raik, S., Durnota, B.: The evolution of sporting strategies. In: Stonier, R.J., Yu, X.H. (eds.) Complex Systems: Mechanisms of Adaption, pp. 85–92. IOS Press, Amsterdam (1994)
13. Soule, T., Heckendorn, R.B.: Evolutionary optimization of cooperative heterogeneous teams. In: SPIE Defense and Security Symposium (2007)
14. Soule, T., Heckendorn, R.B.: Improving performance and cooperation in multiagent systems. In: Proceedings of the Genetic Programming Theory and Practice Workshop (2007)
15. Thomason, R., Soule, T.: Novel ways of improving cooperation and performance in ensemble classifiers. In: Proceedings of the Genetic and Evolutionary Computation Conference: GECCO-2007, pp. 1708–1715. Morgan Kaufmann, San Francisco (2007)

Winning Ant Wars: Evolving a Human-Competitive Game Strategy Using Fitnessless Selection

Wojciech Jaśkowski, Krzysztof Krawiec, and Bartosz Wieloch

Poznan University of Technology, Poznań, Poland
Institute of Computing Science

Abstract. We tell the story of *BrilliAnt*, the winner of the Ant Wars contest organized within GECCO'2007, Genetic and Evolutionary Computation Conference. The task for the Ant Wars contestants was to evolve a controller for a virtual ant that collects food in a square toroidal grid environment in the presence of a competing ant. BrilliAnt, submitted to the contest by our team, has been evolved through competitive one-population coevolution using genetic programming and a novel *fitnessless selection* method. In the paper, we detail the evolutionary setup that lead to BrilliAnt's emergence, assess its human-competitiveness, and describe selected behavioral patterns observed in its strategy.

1 Introduction

Ant Wars was one of the competitions organized within GECCO'2007, Genetic and Evolutionary Computation Conference, in London, England, July 7–12, 2007. The goal was to evolve a controller for a virtual ant that collects food in a square toroidal grid environment in the presence of a competing ant. In a sense, this game is an extension of the so-called Santa-Fe trail task, a popular genetic programming benchmark, to two-player environment.

Ant Wars may be classified as a probabilistic, two-person board game of imperfect information. Each game is played on a 11x11 toroidal board. Before the game starts, 15 pieces of food are randomly distributed over the board and two players (ants) are placed at predetermined board locations. The starting coordinates of ant 1 and ant 2 are $(5, 2)$ and $(5, 8)$, respectively. No piece of food can be located in the starting cells. An ant has a limited field of view – a square neighborhood of size 5x5 centered at its current location, and receives complete information about the states (empty, food, enemy) of all cells within it.

The game lasts for 35 turns per player. In each turn ant moves into one of 8 neighboring cells. Ant 1 moves first. If an ant moves into a cell with food, it scores 1 point and the cell is emptied. If it moves into a cell occupied by the opponent, it kills it: no points are scored, but only the survivor can go on collecting food until the end of the game. Moving into an empty cell has no extra effect. A game is won by the ant that attains higher score. In case of tie, Ant 1 is the winner.

As the game outcome strongly depends on food distribution, the games may be grouped into *matches* played on different boards. Each match consists of $2 \times k$

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 13–24, 2008.
© Springer-Verlag Berlin Heidelberg 2008

games played on k random boards generated independently for each match. To provide for fair play, the contestants play two games on the same board, in the first game taking roles of Ant 1 and Ant 2, and then exchanging these roles; we refer to such a pair of games a *double-game*. To win the match, an ant has to win $k+1$ or more games within the match. In the case of tie, the total score determines the match outcome. If there is still a tie, a randomly selected contestant wins.

The Ant War contestants were required to produce an ANSI-C function $Move(grid, row, column)$, where *grid* is a two-dimensional array representing board state, and (*row*, *column*) represents ant's position. The function was supposed to indicate ant's next move by returning direction encoded as an integer from interval $[0, 7]$. Function code was limited to 5kB in length.

In this paper, we tell the story of Ant Wars winner, *BrilliAnt*, an ant submitted by our team. BrilliAnt has been evolved through competitive one-population coevolution using genetic programming (GP) and a novel *fitnessless selection* method. Despite being conceptually simpler than fitness-based selection, fitnessless selection produces excellent players without externally provided yardstick, like a human-made strategy. An extensive computational experiment detailed in the paper proves that BrilliAnt and other artificial ants evolved using this approach are highly human-competitive in both direct terms (playing against a human opponent) and indirect terms (playing against a human-devised strategy).

In the following Section 2 we shortly summarize the past game-related research in GP. Section 3 describes the model of board perception and the repertoire of GP functions used for strategy encoding. Section 4 provides details on experimental setup and defines the fitnessless selection method. In Section 5, we assess human-competitiveness of the evolved ants, and in Section 6 we describe the most interesting behavioral patterns observed in BrilliAnt's strategy.

2 Genetic Programming for Evolving Game Players

Achieving human-competitive performance in game playing has been AI's holy grail since its very beginning, when game playing strategies, like the famous Bernstein's chess and Samuel's checker players, were hand-crafted by humans. The most spectacular achievement of AI in the game domain was the grand master Garry Kasparov's defeat in duel with Deep Blue, which implemented a brute force approach supported by human expertise. Through successful, it is dubious whether this kind of approach can be applied to more complicated games, and how much does it help to understand and replicate the human intelligence. The $1.5M Ing Prize for the first computer player to beat a nominated human competitor in the Go game is still untouched, presumably because Go has too many states to be approached by brute force. Hard AI is also often helpless when it comes to real-time (strategy) games [3] or multi-agent games where the number of possible states can be even greater than in Go. Things get more complicated also for hand-designed algorithms when the game state is only partially-observable or the game is probabilistic by nature.

The partial failure of hard AI in devising truly intelligent approach to games clearly indicates that handcrafting a good game-playing strategy for a nontrivial game is a serious challenge. The hope for progress in the field are the methods that automatically construct game playing programs, like genetic programming (GP, [7]) used in our approach.

Koza was the first who used GP to evolve game strategies [6] for a two-person, competitive, simple discreet game. Since then, other researchers have demonstrated that the symbolic nature of GP is suitable for this kind of task. Studies on the topic included both trivial games such as Tic Tac Toe [1] or Spoof [16], as well as more complicated and computationally-demanding games, like poker [14]. Core Wars, a game in which two or more programs compete for the control of the virtual computer, is among the popular benchmark problems for evolutionary computations and one of the best evolved players was created using a μGP [4]. Luke's work [8] on evolving soccer softball team for RoboCup97 competition belongs to the most ambitious applications of GP to game playing, involving complicated environment and teamwork. Recently, Sipper and his coworkers demonstrated [13] human-competitive GP-based solutions in three areas: backgammon [2], RoboCode [12] (tank-fight simulator) and chess endgames [5].

3 Ant's Architecture

In the game of Ant Wars introduced in Section 1, ant's field of view (FOV) contains 25 cells and occupies 20.7% of the board area. The expected number of visible food pieces is 3.02 when the game begins. The probability of having n food pieces within FOV drops quickly as n increases and, for instance, for $n = 8$ amounts to less than 0.5%. This, together with FOV's rotational invariance and symmetry, indicates that the number of unique and realistically possible FOV states is low, and any strategy based on the current (observed) FOV state only cannot be competitive in a long run. More may be gained by virtually extending the FOV, i.e., keeping track of past board states as the ant moves. To enable this, we equip our ants with memory, implemented as three arrays overlaid over the board:

- *Food memory* F, that keeps track of food locations observed in the past,
- *Belief table* B, that describes ant's belief in the current board state,
- *Track table* V, that marks the cells visited by ant.

At each move, we copy food locations from ant's FOV into F. Within FOV, old states of F are overridden by the new ones, while F cells outside the current FOV remain intact. As board states may change subject to opponent's actions and make the memory state obsolete, we simulate memory decay in the belief table B. Initially, the belief for all cells is set to 0. Belief for the cells within FOV is always 1, while outside FOV it decreases exponentially, by 10% with each move. Table V stores ant's 'pheromone track', initially filled with zeros. When ant visits a cell, the corresponding element of V is set to 1.

To evolve our ants, we use tree-based, strongly typed genetic programming. A GP tree is expected to evaluate the utility of the move in a particular direction: the more attractive the move, the greater tree's output. To benefit from rotational invariance, we use one tree to evaluate multiple orientations. However, as ants are allowed to move horizontally, vertically, and diagonally, we evolve two trees in each individual to handle these cases: a '*straight*' tree for handling main directions (N, E, S, W) and a '*diagonal*' tree to handle the diagonal directions (NE, NW, SE, SW)[1]. We present the FOV state to the trees by appropriately rotating the coordinate system by a multiple of 90 degrees; this affects both FOV and the ant's memory. The orientation that maximizes trees' output determines the ant's move; ties are resolved by preferring the earlier maximum.

Our ants use three data types: *float* (F), *boolean* (B), and *area* (A). An area represents a rectangle stored as a quadruple of numbers: midpoint coordinates (relative to ant's current position, modulo board dimensions) and dimensions. In theory, the number of possible values for area type is high, so it would be hard for evolution to find the most useful of them. That it why we allow only for relatively small areas, such that their sum of dimensions does not exceed 6. For instance, the area of dimensions (2, 5) cannot occur in our setup.

The set of GP terminals includes the following operators:

- Const(): Ephemeral random constant (ERC) for type F ($[-1; 1]$),
- ConstInt(): Integer-valued ERC for type F (0..5),
- Rect(): ERC for type A,
- TimeLeft() – the number of moves remaining to the end of the game,
- Points() – the number of food pieces collected so far by the ant,
- PointsLeft() – returns 15−Points().

Functions implementing non-terminal nodes (operators):

- IsFood(A) – returns *true* if the area A contains at least one piece of food,
- IsEnemy(A) – returns *true* if the area A contains the opponent,
- Logic operators: And(B, B), Or(B, B), Not(B),
- Arithmetic comparators: IsSmaller(F, F), IsEqual(F, F),
- Scalar arithmetics: Add(F, F), Sub(F, F), Mul(F, F),
- If(B, F, F) – evaluates and returns second child if first child returns true, otherwise evaluates and returns its third child,
- NFood(A) – the number of food pieces in the area A,
- NEmpty(A) – the number of empty cells in the area A,
- NVisited(A) – the number of cells already visited in the area A,
- FoodHope(A) – returns the estimated number of food pieces that may be reached by the ant within two moves (assuming the first move is made straight ahead, and the next one in arbitrary direction).

[1] We considered using a single tree and mapping diagonal boards into straight ones; however, this leads to significant topological distortions which could possibly significantly deteriorate ant's perception.

Note that functions that take the argument of area type compute their return value basing not only on FOV, but on the food memory table F and the belief table B. For example, NFood(a) returns the scalar product, constrained to area a, of table F (food pieces) and table B (belief).

One should also emphasize that all GP functions mentioned here are straightforward. Even the most complex of them boil down to counting matrix elements in designated rectangular areas. Using more sophisticated functions would be conflicting with contests rules that promoted solutions where the intelligence was evolved rather than designed.

4 How BrilliAnt Evolved

In our evolutionary runs ants undergo competitive evaluation, i.e., face each other rather than an external selection pressure. This is often called one-population coevolution [10] or competitive fitness environment [1,8]. In such environments, the fitness of an individual depends on the results of games played with other individuals from the same population. The most obvious variant of this approach is the *round-robin tournament* that boils down to playing one game between each pair of individuals. The fitness of an individual is defined as the numbers of games won. Since the round-robin tournament needs $n(n-1)/2$ games to be played in each generation for population of size n, some less computationally demanding methods were introduced.

Angeline and Pollack [1] proposed *single-elimination tournament* that requires only $n-1$ games to be played. In each round the players/individuals are paired, play a game, and the winners pass to the next round. At the end, when the last round produces the final winner of the tournament, fitness of each individual is the number of won games. Another method reported in literature, *k-random opponents*, defines individual's fitness as the average result of games with k opponents drawn at random from the current population. The method requires kn games to be played. The special case of this method for $k = 1$ is also known as *random pairing*. An experimental comparison between k-random opponents and single-elimination tournament may be found in [11].

Here we propose a novel selection method called *fitnessless selection*. It does not involve explicit fitness measure and thus renders the evaluation phase of evolutionary algorithm redundant. Fitnessless selection resembles tournament selection, as it also selects the best one from a small set of individuals drawn at random from the population. In the case of tournament selection the best individual is the one with the highest fitness. Since our individuals do not have explicit fitness, in order to select the best, we apply a single-elimination tournament, in which the winner of the last (final) round becomes immediately the result of selection. This feature, called *implicit fitness*, makes our approach significantly different from most of contributions presented in literature. The only related contribution known to us is [15].

Using ECJ [9] as the evolutionary engine, we carried out a series of preliminary experiments with various evolutionary setups, including island model and

different variants of selection procedure. In a typical experiment, we evolved a population of 2000 individuals for 1500 generations, which took approx. 48 hours on a Core Duo 2.0 GHz PC (with two evaluating threads). In all experiments, we used probabilities of crossover, mutation, and ERC mutation, equal to 0.8, 0.1, and 0.1, respectively. GP trees were initialized using ramped half-and-half method, and were not allowed to exceed depth 8. For the remaining parameters, we used ECJ's defaults [9].

We relied on the default implementation of mutation and crossover available in ECJ, while providing specialized ERC mutation operators for particular ERC nodes. For Const() we perturb the ERC with a random, normally distributed value with mean 0.0 and standard deviation 1/3. For ConstInt() we perturb the ERC with a random, uniformly distributed integer value from interval $[-1; 1]$. For Rect() we perturb each rectangle coordinate or dimension with a random, uniformly distributed integer value from interval $[-1; 1]$. In all cases, we trim the perturbed values to domain intervals.

To speed up the selection process and to meet contest rules that required the ant code to be provided in C programming language (ECJ is written in Java), in each generation we serialize the entire population into one large text file, encoding each individual as a separate C function with a unique name. The resulting file is then compiled and linked with the game engine, which subsequently carries out the selection process, returning the identifiers of selected individuals to ECJ. As all individuals are encoded in one C file, the compilation overhead is reasonably small, and it is paid off by the speedup provided by C (compared to Java). This approach allows us also to monitor the actual size of C code, constrained by contest rules to 5kB per individual.

The best evolved ant, called *BrilliAnt* in the following, emerged in an experiment with population of 2250 individuals evolving for 1350 generations, using fitnessless selection with tournament size 5 (thus 4 matches per single-elimination tournament), and with 2×6 games played in each match. BrilliAnt has been submitted to GECCO'07 Ant Wars competition and won it. We would like to point out that BrilliAnt evolved and was selected in completely autonomous way, without support from any human-made opponent. To choose it, we ran a round-robin tournament between all 2250 individuals from the last generation of the evolutionary run. It is worth noticing that this process was computationally demanding: having only one double-game per match, the total number of games needed was more than 5,000,000, i.e., as much as for about 47 generations of evolution.

5 Human Competitiveness

The game-playing task allows for two interpretations of human competitiveness. To assess the *direct competitiveness* we implemented a simulator that allows humans to play games against an evolved ant. Using this tool, an experienced human player played 150 games against BrilliAnt, winning only 64 (43%) of them and losing the remaining 86 (57%). BrilliAnt's total score amounted to 1079, compared to human's 992. Even when we take into account the fact, that playing

Table 1. The results of a round-robin tournament between the evolved ants (in bold) and humants (plain font). Each match consisted of $2 \times 100,000$ games.

Player	Matches won	Games won	Total score
ExpertAnt	6	760,669	10,598,317
HyperHumant	6	754,303	10,390,659
BrilliAnt	6	753,212	10,714,050
EvolAnt3	3	736,862	10,621,773
SuperHumant	3	725,269	10,130,664
EvolAnt2	3	721,856	10,433,165
EvolAnt1	1	699,320	10,355,044
SmartHumant	0	448,509	9,198,296

150 games in a row may be tiring for a human and cause him/her make mistakes, this result can be definitely considered as human competitive. The reader is encouraged to measure swords with BrilliAnt using Web interface provided at http://www.cs.put.poznan.pl/kkrawiec/antwars/.

We analyzed also *indirect competitiveness*, meant as ant's performance when playing against human-designed *programs* (strategies), called *humants* in the following. We manually implemented several humants of increasing sophistication and compared them with the evolved ants using matches of $2 \times 100,000$ games. Let us emphasize that the C programming language used for that purpose offers richer control flow (e.g., loops) and more arbitrary access to game board than the GP encoding, so this gives a significant handicap to humants. Nevertheless, the first of our humants was easily beaten by an ant evolved in a preliminary evolutionary run that lasted 1000 generations with GP tree depth limit set to 7. The next one, *SmartHumant*, seemed more powerful until we increased the depth limit to 8 and equipped ant with memory. That resulted in evolving an ant that beats even SmartHumant. Having learned our lessons, we finally designed *SuperHumant* and *HyperHumant*, the latter being the best humant we could develop. HyperHumant stores states of board cells observed in the past, plans 5 moves ahead, uses a probabilistic memory model and several end-game rules (e.g., *when your score is 7, eat the food piece even if the opponent is next to it*).

To our surprise, by tuning some evolutionary operators we were able to evolve an ant, *ExpertAnt*, that wins 50.12% of games against HyperHumant. The difference in the number of games won between ExpertAnt and HyperHumant is statistically insignificant at the typical 0.01 level, but it is significant at the 0.15 level. As BrilliAnt turned out to be a bit worse than HyperHumant (loosing 52.02% of games), ExpertAnt apparently could be considered a better pick for the Ant Wars contest. However, although ExpertAnt evolved without human intervention, it has been selected by explicitly testing all ants from the last generation against the *manually designed* HyperHumant. As our intention was to evolve contestant fully autonomously, so, notwithstanding ExpertAnt performance, we decided to submit BrilliAnt to the contest as it evolved and has been selected completely autonomously. Quite interestingly, we observed also that the

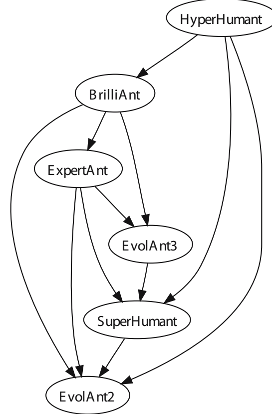


Fig. 1. Graph showing relations between players. An arrow leading from ant a to ant b means that a is statistically better than b ($\alpha = 0.01$). $2 \times 100,000$ games were played between every two ants. EvolAnt1 and SmartHumant were not showed to improve graph's readability. EvolAnt1 wins against SmartHumant only.

method used to select ExpertAnt probably promotes overfitting: despite being slightly better than HyperHumant, ExpertAnt loses against BrilliAnt (in 51.77% of games).

Table 1 presents the results of a round-robin tournament between eight ants, the five mentioned earlier and three other evolved ants (*EvolAnt**). Each participant of this contest played 1,400,000 games against seven competitors and could maximally score 21,000,000. It is hard to say which ant is the ultimate winner of this tournament. Three of them won six matches each. ExpertAnt won the most games, but it is BrilliAnt that got the highest total score.

The results of the same experiment are shown also in the form of graph in Fig. 1. An arrow leading from a to b indicates that a turned out to be statistically better than b (at 0.01 level). No arrows between ants means no statistical advantage. HyperHumant is the only player that never loses significantly and in this respect it can be considered as the winner of the tournament. Interestingly, there are no cycles in this graph and it is weakly transitive.

6 BrilliAnt's Strategy

As BrilliAnt's code is too complex to analyse it within this paper, we describe selected observations concerning its behavior. Let us start from the most obvious strategies. Faced with two corner areas of the field of view (FOV) occupied by food, BrilliAnt always selects the direction that gives chance for more food pieces. It also reasonably handles the trade-off between food amount and food proximity, measured using chessboard (Chebyshev) distance (the number of moves required to reach a board cell). For instance, given a group of two pieces of food at distance

2 ((2, 2) for short), and a group of two pieces of food in distance 1, i.e., (2, 1), BrilliAnt chooses the latter option, a fact that we shortly denote as $(2, 2) \prec (2, 1)$. Similarly, $(1, 1) \prec (2, 2)$, $(3, 2) \prec (2, 1)$, $(3, 2) \prec (3, 1)$, and $(2, 2) \prec (3, 2)$. If both groups contain the same number of food pieces but one of them is accompanied by the opponent, BrilliAnt chooses the other group. It also makes reasonable use of memory: after consuming the preferred group of food pieces, it returns to the other group, unless it has spotted some other food in the meantime.

Food pieces sometimes happen to arrange into 'trails', similar to those found in the Artificial Ant benchmarks [7]. BrilliAnt perfectly follows such paths as long as the gaps are no longer than 2 cells (see Fig. 2). However, when faced with a large group of food pieces, it not always consumes them in an optimal order.

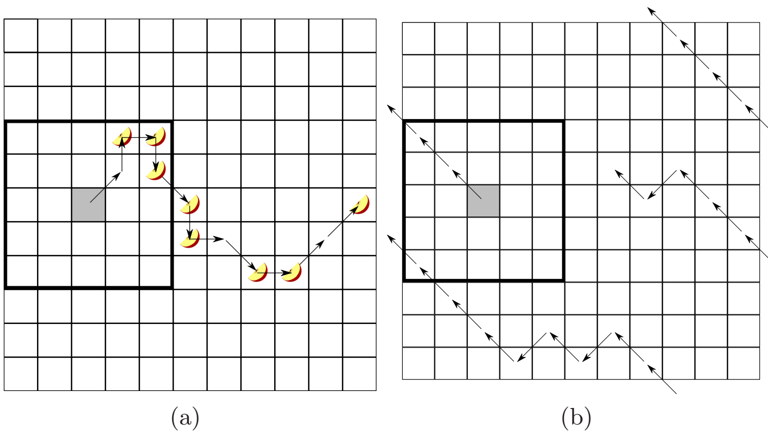
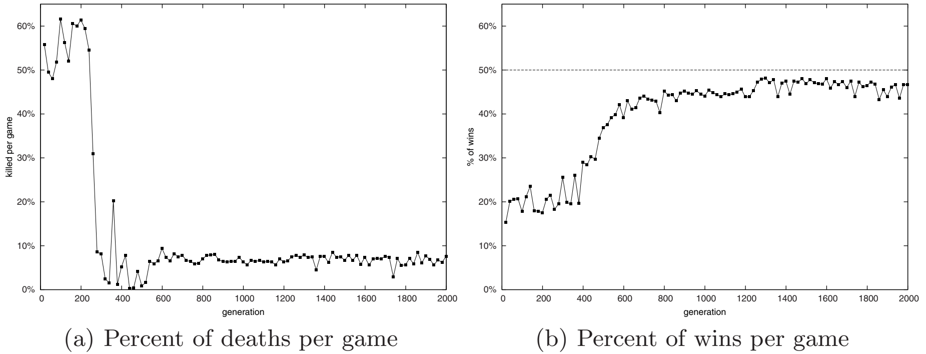


Fig. 2. Brilliant's behaviors when following a trail of food pieces (a), and in absence of food (b). Gray cell and large rectangle mark Brilliant's starting position and initial FOV, respectively.

If the FOV does not contain any food, BrilliAnt proceeds in the NW direction. However, as the board is toroidal, keeping moving in the same direction makes sense only to a certain point, because it brings the player back to the starting point after 11 steps, with a significant part of the board still left unexplored. Apparently, evolution discovered this fact: after 7 steps in the NW direction (i.e., when FOV starts to intersect with the initial FOV), BrilliAnt changes direction to SW, so that the initial sequence of moves is: 7NW, 1SW, 1NW, 1SW, 6NW, 1SW, 1NW. A simple analysis reveals that this sequence of 18 moves, shown in Fig. 2b, provides the complete coverage of the board. This behavior seems quite effective, as the minimal number of moves that scans the entire board is 15. Note also that in this sequence BrilliAnt moves only diagonally. In absence of any other incentives, this is a locally optimal choice, as each diagonal move uncovers 9 board cells, while a non-diagonal one uncovers only 5 of them.

Evolving this full-board scan is quite an achievement, as it manifests in complete absence of food, a situation that is close to impossible in Ant Wars, except for the highly unlikely event of the opponent consuming all the food earlier. BrilliAnt exhibits variants of this behavioral pattern also *after* all some food pieces have been eaten and its FOV is empty.

BrilliAnt usually avoids the opponent, unless it comes together with food and no other food pieces are in view. In such a case, it cannot resist the temptation and approaches the food, maintaining at least distance 2 from the opponent. For one food piece, this often ends in a deadlock: the players hesitatingly walk in the direct neighborhood of the food piece, keeping safe distance from each other. None of them can eat the piece, as the opponent immediately kills such a daredevil. However, there is one exception from this rule: when the end of game comes close and the likelihood of finding more food becomes low, it may pay off to sacrifice one's life in exchange for food. This in particular applies to the scenario when both players scored 7 and the food piece of argument is the only one left.



(a) Percent of deaths per game (b) Percent of wins per game

Fig. 3. Graphs show evolution dynamics for a typical process of evolution. Each point corresponds to an best-of-generation ant chosen on the basis of 2×250 games against HyperHumant. The presented values are averaged over 2×10000 games against HyperHumant. It can be noticed that the evolution process usually converges around 1300 generation when the wining rate against a fixed opponent ceases to improve.

This sophisticated 'kamikaze' behavior evolved as a part of BrilliAnt's strategy and emerged also in other evolutionary runs. Figure 3b illustrates this behavior in terms of death rate statistic for one of the experiments. The ants from several initial generations play poorly and are likely to be killed by the opponent. With time, they learn how to avoid the enemy and, usually at 200-300th generation, the best ants become perfect at escaping that threat (see Fig. 3b). Then, around 400-500th generation, the ants discover the benefit of the 'kamikaze' strategy, which results in a notable increase of death rate, but pays off in terms of winning frequency.

7 Conclusions

This paper presented an evolved game strategy that won the Ant Wars contest and has been produced by means of a novel fitnessless mechanism of selection. This mechanism lets individuals play games against each other and simply propagates the winner to the next generation, allowing us to get rid of the objective fitness. Though unusual from the viewpoint of the core EC research, selection without fitness has some rationale. The traditional fitness function used in EC is essentially a mere technical means to impose the selective pressure on the evolving population. It is often the case that, for a particular problem, the definition of fitness is artificial and usually does not strictly conform its biological counterpart, i.e., the *a posteriori* probability of the genotype survival. By eliminating this need, we avoid subjectivity that the fitness definition is prone to.

Despite its simplicity, the evolution with fitnessless selection produces sophisticated human-competitive strategies. We do not entice the evolution by providing competitive external (e.g., human-made) opponents, so that both evolution as well as selection of the best individual from the last generation are completely autonomous. Improvement of individuals' performance takes place only thanks to competition between them. Let us also emphasize that these encouraging results have been obtained despite the fact that the game itself is not trivial, mainly due to incompleteness of information about the board state available to the players.

Interestingly, in our evolutionary runs we have not observed any of the infamous pathologies common to coevolution, like loss of gradient or cycling. This may be probably attributed to the fact that our setup involves single a population. The detailed comparison of the fitnessless selection and fitness-based selection methods will be subject of a separate study.

So, is it really true that an evolved solution can be better than human's mind? Check at the page http://www.cs.put.poznan.pl/kkrawiec/antwars/ if you can beat BrilliAnt!

Acknowledgment

This research has been supported by the Ministry of Science and Higher Education grant # N N519 3505 33.

References

1. Angeline, P.J., Pollack, J.B.: Competitive environments evolve better solutions for complex tasks. In: Forrest, S. (ed.) Proceedings of the 5th International Conference on Genetic Algorithms, pp. 264–270 (1993)
2. Azaria, Y., Sipper, M.: GP-gammon: Genetically programming backgammon players. Genetic Programming and Evolvable Machines 6(3), 283–300 (2005)
3. Buro, M.: Real-time strategy games: A new AI research challenge. In: Gottlob, G., Walsh, T. (eds.) IJCIA, pp. 1534–1535. Morgan Kaufmann, San Francisco (2003)

4. Corno, F., Sanchez, E., Squillero, G.: On the evolution of corewar warriors. In: Proceedings of the 2004 IEEE Congress on Evolutionary Computation, June 20-23, 2004, pp. 133–138. IEEE Press, Los Alamitos (2004)
5. Hauptman, A., Sipper, M.: Evolution of an efficient search algorithm for the mate-in-N problem in chess. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 78–89. Springer, Heidelberg (2007)
6. Koza, J.R.: Genetic evolution and co-evolution of game strategies. In: The International Conference on Game Theory and Its Applications, Stony Brook (1992)
7. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
8. Luke, S.: Genetic programming produced competitive soccer softbot teams for robocup97. In: J.R.K., et al., (eds.) Genetic Programming 1998: Proceedings of the 3rd Annual Conference, Madison, Wisconsin, USA, pp. 214–222 (1998)
9. Luke, S.: ECJ evolutionary computation system (2002), http://cs.gmu.edu/eclab/projects/ecj/
10. Luke, S., Wiegand, R.: When coevolutionary algorithms exhibit evolutionary dynamics. In: 2002 Genetic and Evolutionary Computation Conference Workshop Program, pp. 236–241 (2002)
11. Panait, L., Luke, S.: A comparison of two competitive fitness functions. In: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 503–511. Morgan Kaufmann, San Francisco (2002)
12. Shichel, Y., Ziserman, E., Sipper, M.: GP-robocode: Using genetic programming to evolve robocode players. In: Keijzer, M., Tettamanzi, A.G.B., Collet, P., van Hemert, J.I., Tomassini, M. (eds.) EuroGP 2005. LNCS, vol. 3447, pp. 143–154. Springer, Heidelberg (2005)
13. Sipper, M.: Attaining human-competitive game playing with genetic programming. In: El Yacoubi, S., Chopard, B., Bandini, S. (eds.) ACRI 2006. LNCS, vol. 4173, Springer, Heidelberg (2006)
14. Smilak, K.C.: Finding the ultimate video poker player using genetic programming. In: Koza, J.R. (ed.) Genetic Algorithms and Genetic Programming at Stanford 1999, pp. 209–217 (1999)
15. Tettamanzi, A.G.B.: Genetic programming without fitness. In: Koza, J.R. (ed.) Late Breaking Papers at the Genetic Programming 1996 Conference (1996)
16. Wittkamp, M., Barone, L.: Evolving adaptive play for the game of spoof using genetic programming. In: S.J.L., et al. (eds.) Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games (CIG 2006), University of Nevada, Reno, USA, pp. 164–172. IEEE, Los Alamitos (2006)

In Silicon No One Can Hear You Scream: Evolving Fighting Creatures

Thomas Miconi

School of Computer Science,
University of Birmingham,
Birmingham B152TT, UK
txm@cs.bham.ac.uk

Abstract. Virtual creatures operating in a physically realistic 3D environment, as originally introduced by Karl Sims, provide a challenging domain for artificial evolution. However, few coevolutionary experiments have been reported. Here we describe the results of our experiments on the evolution of physical combat among virtual creatures: essentially, we evolve creatures that trade blows with each other. While several authors have involved highly abstract forms of "combat" in their systems, this is (to our knowledge) the first example of realistic physical combat between virtual creatures, based on actual contact and physical damage. This poses the question of apportioning damage in a collision. Our solution is to assign damage proportionally to how much each colliding limb contributed to the occurrence and depth of the collision. Our system successfully evolves a wide range of morphologies and fighting behaviours, which we describe visually and verbally. As with our previous efforts, our source code is publicly available.

1 Introduction

1.1 Virtual Creatures

More than a decade ago, Karl Sims presented the results of his experiments on the evolution of virtual creatures in a three-dimensional (3D), physically realistic environment [1,2]. Virtual creatures offer a potentially boundless ground for evolutionary experimentation. The complexity of physical interactions between 3D structures creates a challenging task for evolution, providing an ideal test-bed for evolutionary algorithms and techniques. In addition, there are immediate practical applications to evolving virtual creatures, such as modular robotics [3,4] or self-modelling in robots. [5]

While there has been a significant amount of work in projects related to the simulation of 3D creatures, initially, much of it was concerned with specific areas of research, such as gene regulation in development [6] or modular robotics [4,3]. Other authors built environments based on simplified physics, such as Hornby & Pollack [7] or the GOLEM project [8]. The Framsticks project [9] uses stick-figure creatures and allows users to build simulations through scripts.

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 25–36, 2008.
© Springer-Verlag Berlin Heidelberg 2008

Reproductions of Sims' results were a long time coming, owing no doubt to the lack of affordable hardware and software resources. Increases in computational power, as well as the emergence of widely available physics simulation libraries, have made it easier to undertake such projects in recent years. After an early attempt at a partial replication by Taylor and Massey [10], we described the first complete replication (and extension) of Sims' results, using standard McCulloch-Pitts neurons rather than the set of complex functional neurons used by Sims [11]. Chaumont and colleagues [12] reimplemented Sims' model and successfully applied it to the evolution of catapults. Shim and Kim [13] evolved flying creatures, although with simplified controllers (sinusoidal functions rather than neural networks) and more constrained morphologies. Lassabe and colleagues [14] also implemented a Sims-like system, using classifier systems selecting among pre-set activation patterns rather than neural networks, and used it to evolve various locomotive behaviours in rugged environments (including relief, trenches, etc.) and simple tasks such as block-pushing. Simultaneously, Bongard and colleagues [5] have explored new directions in the joint evolution of morphology and behaviour: actual robots in the real world engage in continuous self-modelling and self-simulation, in effect evolving models of themselves. This allows the robot to recover from random damage, e.g.: "when a leg part is removed, [the robot] adapts the self-models, leading to the generation of alternative gaits."

2 Evolving Fighting Creatures

2.1 Coevolution: The "box-grabbing" Problem and Its Limitations

Sims' original paper on coevolution [2] was based on the simple task of grabbing a small cube away from an opponent. Creatures are positioned on opposite sides and at equal distances from a cubic box (with corrections for their height), and left to act for a fixed period of time. The final score for each creature is the normalised difference between this creature's distance to the box and its opponent's distance to the box.

The box-grabbing task has many advantages, not least simplicity: it is easy to understand, easy to evaluate numerically, and easy to implement. It also has the less obvious advantage of offering a fitness function that can "work" at all stages of the evolutionary process, in that it can offer an informative evaluation both to very poor and very advanced competitors. This is due to the fact that it is based on relative distances, and that even the most primitive creatures will possess some heritable variance in this characteristic (if only by falling down).

However, this simplicity can also be seen as a limitation. While there are several ways to grab a box, the variety of efficient behaviours is necessarily limited. Another problem is that it is not easy to see how this task could be extended to large numbers of competing individuals. We might imagine box-grabbing competitions involving a few creatures; we might even fancy the evolution of "rugby-playing" creatures, in which teams of individuals would compete against each other. But there does not seem to be any obvious way in which box-grabbing could meaningfully be used in an open environment involving many independent

individuals, constantly competing against each other, with varying lifespans and asynchronous births and eliminations.

2.2 Physical Combat: The Appeal of (Virtual) Violence

Physical combat between creatures appears intuitively appealing as a basis for evolution. This comes in no small part from the fact that physical combat is ubiquitous in nature. Predation, sexual competition among males and other forms of fighting have been fruitful sources of evolutionary creativity in many lineages, producing remarkable examples of arms races and mutual adaptations.

Another attractive feature of physical combat is that it is a very direct form of interaction, requiring no mediating device or instrument (as opposed to box-grabbing, and therefore box-requiring, experiments). This means that it can be used in many different settings with relatively few constraints. Thus physical combat could be used in an open environment in which a population of individuals would interact and evolve freely, in an unsupervised fashion.

2.3 Related Work

Many evolutionary experiments use some idealised form of "fighting" or "killing" behaviour as part of a range of pre-defined behaviours. These include Geb [15], Echo [16], Polyworld [17], Framsticks [9] and others. However, in these systems, the actual process of fighting is essentially *abstract*. It corresponds to a pre-defined rule, hard-coded into the program, such as "eliminate the individual with lowest energy level," or even simply "eliminate the individual right in front of you, no matter what" (as in Geb). Evolution bears on when and how to use the abstract fighting behaviour, not on how to fight.

In fact, despite the possibilities offered by physical combat, we have only been able to find one published attempt at evolving physical combat in a 3D environment: O'Kelly and Hsiao [18] have implemented a modified version of Sims' model, based on a very simple form of combat. In this system, "the first creature to touch its enemy's root node is deemed the winner." This simplified form of combat is easy to implement and assess, and avoids the difficulties described in the following sections. However, it is also less flexible in many ways, not least in being an "all-or-nothing" measure of success. To provide a gradient for evolution, O'Kelly and Hsiao add another component to their fitness function: at the end of each round, both creatures are rewarded with a value inversely proportional to the final distance between the two. This is expected to favour the emergence of simple approach behaviours in the early stages of evolution. Of course this has the drawback that the corresponding reward is equally given to both creatures, independently of how much each creature contributed to reducing this distance.[1]

[1] A simple way to reward creatures more fairly would be to calculate, at each timestep, the modification in the distance between the position of each creature and the *previous* position of the other. In this manner, creatures that actually move towards their opponent could be rewarded, while those which stay put or move away from their opponents would not.

Another problem with this method of combat, especially for our own block-based creatures, is that it has an obvious weak point: simply protecting the root limb makes a creature effectively invincible.

We would like to create a more realistic system, relying on a less abstract form of combat. Instead, we would like to evolve *actual* physical fight, based on physical shock, very much as in the real world. In such a system, a fighter's success would depend on how much physical damage it has inflicted upon (and received from) its opponent. Basically, what we seek is a system in which creatures would evolve to literally beat each other up. To our knowledge, no such system has been reported in the literature.

2.4 Difficulties of Physical Combat: Newton vs. Darwin

The central question in physical combat is to determine how damage should be evaluated: when do we say that an individual has somehow hurt, or otherwise dominated, its opponent? This apparently simple question turns out to pose significant problems.

The most obvious answer is simply to use impacts (and some measure of kinetic energy at the time of impact) as the basis of combat: essentially, to let individuals trade blows with each other. However, this introduces a difficulty caused by Newton's third law (often summarised as "action equals reaction"). If two rigid blocks come into collision, and suffer some damage as a result, then both blocks will suffer *equivalent* damage. This is because physical damage is mostly related to kinetic energy. Clearly the relative velocities of each limb with regard to the other are equal in magnitude (and of opposite signs), and the resulting kinetic energy (and associated impact damage) will therefore be equal for both. The consequence is that when a creature hits another, the creature dealing the blow will suffer the *same* damage as the one receiving it. Clearly this is not conducive to the evolution of fighting behaviours.

In nature, the main reason why physical combat can occur is simply the heterogeneity of materials. Flesh, bones, teeth, skin, horn, etc., have different properties that make it possible to inflict damage on an opponent without suffering too much as a result. The cheetah's claws are harder than the gazelle's skin and flesh, and can therefore damage it more than they are damaged by it. Martial arts fighters attempt to throw their fists and heels at their opponent's face and stomach - rather than the other way round - because the bone structure of those parts favour (closed) hands and feet in collisions against the nose and the belly. Additionally, the geometry of object plays a roles: sharp, pointy objects will behave differently than flat or dull objects in collisions - hence the variety of mammalian tooth shapes.

Implementing such variety of materials in our simulation would clearly be cumbersome and difficult to "get right." In addition, we would need to impose some cost on the toughness of materials, to prevent evolution from turning into a simple maximisation of toughness. In nature, such runaway escalation in armour is simply prevented by the trade-offs imposed by available resources and other tasks. This would not be readily transposable in our simple model.

2.5 Solution: Favouring the Aggressor

To overcome this difficulty, we chose to evaluate the damage inflicted by a creature upon another by measuring "how much" this creature contributed to the occurrence and intensity of the collision. The result is that the creature that initiates contact more than the other (that is, the creature that is "dealing the blow") is favoured in the interaction.

Collision intensity is estimated by penetration depth. How can we measure how much each of the colliding limbs contributed to this collision? This is estimated by suspending the simulation, and then letting each of the colliding block in turn move for one timestep at its current velocity, while the other one is kept fixed; the resulting *increase* in penetration depth, if any, is used as a measurement of how much this creature contributed to the collision - that is, how much it actually moved towards the other (see Figure 1). After this, all blocks return to their original positions, and the simulation proceeds normally.

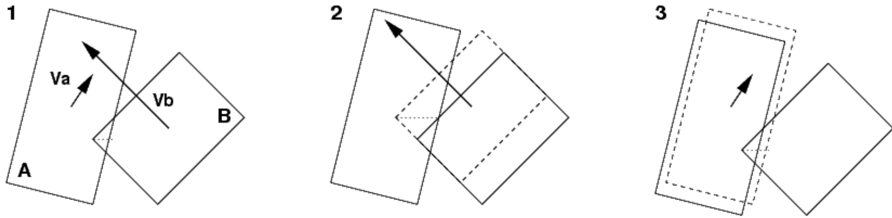


Fig. 1. Damage calculation. 1: A collision occurs between limbs A and B, moving with velocities `Va` and `Vb` respectively. 2: Letting B move at its current velocity for one timestep (while keeping A fixed) results in a large increase in penetration depth. 3: By contrast, letting A move at its current velocity for one timestep (while keeping B fixed) results in a smaller increase in penetration depth. Thus, in this collision, B inflicts more damage upon A than A upon B. Note that if `Va` was pointing away from B, then letting A move for one timestep would actually *reduce* penetration depth, and thus A would not be inflicting any damage upon B at all.

3 System Description

3.1 Virtual Creatures

Our system has already been described in previous publications (e.g. [19,11]). The system used here is very similar, with minor differences. Here we only provide a brief overview of the platform, including differences with previously published material. As with our previous efforts, the source code of our experiments is freely available (together with pictures and videos) at the following URL:
`http://www.cs.bham.ac.uk/~txm/creatures/`

Morphology: As in Sims' model, the creatures are branching structures composed of rigid 3D blocks. Each block (or "limb") is connected to its parent limb by a hinge joint, except for the first ("root" or "trunk") limb which obviously

has no parent. Hinge joints have limited amplitude, so that rotation can only occur within the $[-3\pi/4, 3\pi/4]$ range. The genetic specification of a creature is given as a tree of nodes. Each of these nodes contain morphological and neural information about one limb. The morphological information in each genetic node specifies the dimensions of the limb (width, length and height), the orientation of this limb with regard to its parent (in the form of two parameters indicating polar angles with the xz and the xy planes, that is longitude and latitude, in the frame of reference of the parent limb), the direction of movement which may be either "vertical" or "horizontal" (that is aligned either with the y or with the z axis of the limb), and a boolean flag for reflection which governs symmetric replication along the xz plane of its parent. A limb also contains neural information, as described in the following paragraphs.

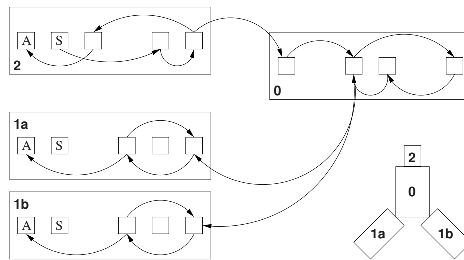


Fig. 2. Organisation of a fictional creature pictured in the bottom-right corner. Limb 0 has no sensor (S) or actuator (A). Limb 1 is reflected into two symmetric limbs 1a and 1b, which share the same morphological and neural information.

Creature control and neural organisation: Our creatures are controlled by neural networks. Each limb may contain up to 5 neurons. Genetic information about a given neuron specifies the activation function for this neuron, a threshold/bias parameter θ, and connection information. The activation function may be either a sigmoid ($\frac{1}{1+\exp^{-(\sigma+\theta)}}$) or the hyperbolic tangent $\tanh(\sigma + \theta)$ where σ is the weighted sum of inputs; the difference between sigmoid and tanh is that the first has values in $[0, 1]$ while the latter has values in $[-1, 1]$. Connection information specifies, for each connection, the source of this connection (that is the neuron whose output is received through this connection) and a weight value. As in Sims' model, neurons can only be connected with other neurons from the same limb, from adjacent limbs, or from the root limb. Each neuron may receive up to 3 connections.

Sensor neurons and actuator neurons are handled specially. The first type of sensor neuron is a proprioceptive neuron, which measures the current angle formed by the hinge joint to which this neuron's limb is attached, scaled within the $[-1, 1]$ range. Additionally, there are "vision" sensors, similar to those used by Sims: these sensors return the distance, along either the x or y axis of the

limb's frame of reference, to the centre of mass of the closest neighbouring animat's trunk limb. Finally, there are contact sensors, the output of which is one if the limb is currently in contact with a limb of another creature, and zero otherwise. Every limb has exactly one proprioceptor, and may have any number of other sensors (within the maximum number of neurons for each limb). In addition, the trunk limb always contains one x sensor and one y sensor.

Actuator neurons command the movement of each limb, that is, its rotation around its joint. The output of an actuator indicates the desired angular velocity around this joint (remember that the joints have limited amplitude). Actuator inputs are defined similarly as other neurons, but their activation function is always a scaled hyperbolic tangent of the form $\tanh(\sigma + threshold)$. Each limb has exactly one actuator.

Expression of the genome: The creatures are constructed according to the information contained in the genetic nodes. A very simple developmental system translates the genotype into a corresponding phenotype, and may introduce additional complexity if the genetic information dictates it. Our system uses the same developmental features as Sims, with some refinements. The first developmental process is *reflection* of limbs: if a limb has its reflection flag set, a symmetric copy of this limb and of all its attached sub-limbs will also be generated, where symmetry is taken along the parent limb's xz (longitudinal) plane. This process allows for bilateral symmetry in the system. Another developmental feature is *recursion*, which effectively models segmentation in biological organisms: each limb may specify a recursion index r, which means that r copies of this limb (and of its sub-limbs) will be sequentially attached to each other, similar to repetitive segments in living animals such as arthropods and vertebrates. A limb may also carry a "terminal" flag, which indicates that, if its parent is recursively replicated, this limb would only be added to the very last instance of the replicated parent. We provide fine-grained control of neural wiring among replicated limbs, allowing for asymmetric information flow between replicated structures, an improvement over Sims' original model.

Genetic operators: We use three genetic operators, broadly similar to those used by Sims. *Crossover* is performed by simply aligning the genetic nodes of both parents in two rows, then building a new list of genetic nodes by concatenating the left part of one parent with the right part of the other. *Grafting* corresponds to the removal of a branch (that is a limb and all its sub-limbs), and its replacement by a branch taken from another individual. Connectivity information is adapted and maintained: the neurons of the trunk establish the same connections with the new branch as they had with the old one, and similarly the new branch has the same connection with its new trunk as it had with its previous trunk. *Mutation* occurs by sequentially and randomly altering each morphological and neural parameter within a genome (from limb size to connection weight) with a given probability P_{mut}, as well as by removing a limb with probability P_{mut} and adding a new, randomly generated limb, also with probability P_{mut}.

3.2 Rules of Engagement

Competitions between two creatures are organised as follows: first, creatures are put on each side of a vertical plane, and then pushed away from each other by a very small distance to avoid any contact. Then creatures are allowed to move according to their controllers' output. Over the first 10% of evaluation time, creatures benefit from an *immunity period*, during which they can neither hurt nor be hurt by each other. After this immunity period has elapsed, damage is evaluated according to the previously described method, and accumulated over the entire evaluation period.

The fact that creatures are initially close favours the probability of contact occurring, even in the very early stages. This provides an immediately exploitable gradient for natural selection to act upon.

At the end of the evaluation period, each creature is given a final score equal to 1 + (Damage inflicted - Damage suffered) / (Damage inflicted + Damage suffered). This calculation is inspired by Sims [2]. Note that this score always falls within the $[0, 2]$ range.

4 Experiments and Results

The algorithm we use is a modification of Sims' original algorithm [2], later called "Last Elite Opponent" (LEO) by Cliff & Miller [20]. Following Sims, we use two populations. In essence, Sims' LEO algorithm evaluates individual by making them compete against the current "champion" of the opposing population. At each generation, every member of population 1 competes against the current "champion" of opposing population 2, resulting in a certain score: this score is the fitness of the individual. The 20% highest-scoring individuals are chosen as survivors for the next generation, and the remainder of population 1 is filled with offspring of these survivors; the parents of each new individual are selected from among the survivors via roulette-wheel selection. The highest-scoring individual is also identified as the new "champion" of population 1. Then the same process is applied to population 2: each individual in population 2 competes against the current champion of population 1, a champion is identified based on this score, highest-scoring survivors are selected and the population is filled with offspring of the survivors. This concludes one generation of the algorithm. The cycle is then repeated for as many generations as required. In the first generation, current champions are chosen randomly or arbitrarily.

We modified the LEO algorithm by incorporating a "sliding archive" of past champions in the evaluation process. At every generation, we maintain an archive in which we store the previous champions of each population over the last 15 generations. We make each individual compete, not only against the current opposing champions, but also against a sample of 2 past opposing champions picked from this sliding archive (this sample is randomly selected for each population at the beginning of each generation, so at every generation every individual of each population competes against the same set of opponents). This modification

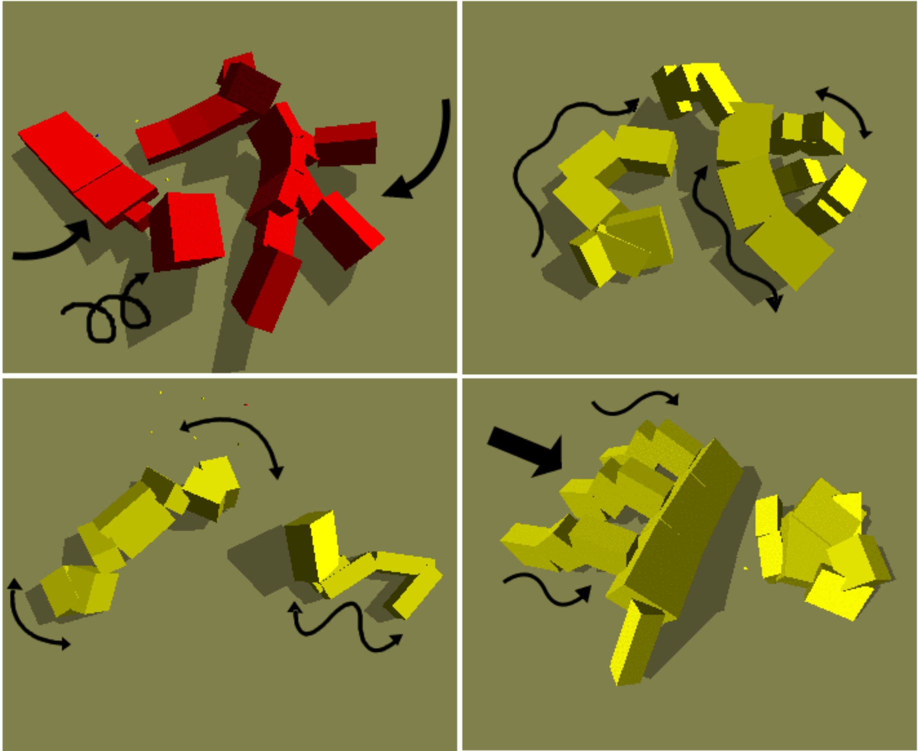


Fig. 3. Four pairs of fighters obtained in the course of the experiments described in the next chapter. In the top-left corner, one simple creature uses its rotating cubic head to perform a "compass" motion, while the other creature uses three rotating appendages both as flails and legs. The dark colour indicates that the creatures are still within their immune period. In the top-right corner, a linear individual constantly aims its wagging tail at its more complex opponent, which uses sensors from its head to coordinate its own movement (the neural network of the larger creature is described in Figure 4). In the bottom-left corner, a two-armed crawler and a directed snake move towards each other. In the bottom-right corner, a large creature uses three undulating appendages as powerful legs to "steamroll" its opponent.

improved the performance of the algorithm, as ascertained by systematically pitting individuals evolved with and without sliding archives against each other.

Useful creatures consistently evolved within a couple of generations. The system generated a wide range of morphologies, as shown in Figure 3. Various strategies emerged, some of which made use of external sensors, while others did not. All non-trivial individuals made use of proprioceptors to synchronise oscillating groups of limbs.

One commonly observed strategy that did not make use of external sensors was the "compass" method: one extremity of the creature remains fixed on the ground (mostly through sheer mass) while the other extremity features a "head"

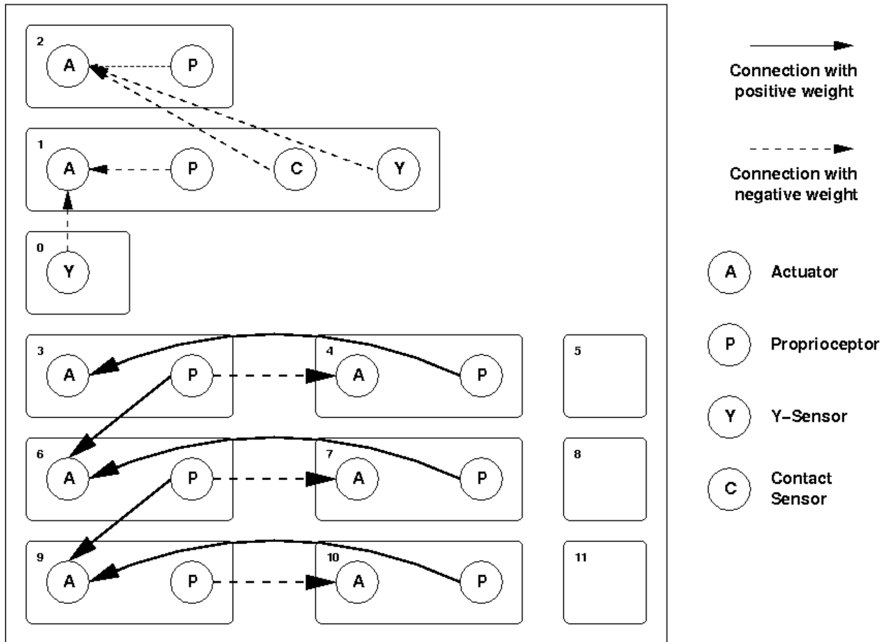


Fig. 4. Neural network of the larger creature in the top-right picture in Figure 3. Each rounded rectangle indicates a limb. Limb 0 corresponds to the "neck" of the individual; limbs 1 and 2 constitute the "head", while the bottom limbs (3-11) represent three replicated segments, each composed of three limbs. Limbs 5, 8 and 11 have no neurons at all and are simply fixed appendages of limbs 4, 7 and 10, respectively. Notice the mutual connections between the proprioceptors and actuators of various limbs, which induce synchronisation between the motions of these limbs: for example, the three repeated segments move in an undulating fashion due to the pattern of direct and indirect connections between the proprioceptors and actuators of successive limbs. This creates a locomotive behaviour, which is guided by the sensor neurons located in the "head".

endowed with a constantly rotating structure that propels this head against the ground. As the head is pushed sideways by the rotating structure, while the tail remains fixed, the creature undergoes a compass-like motion, sweeping its immediate vicinity. In addition, the head's rotating appendage serves as a striking implement to inflict damage upon opponents. This simple strategy proves very effective, as the creature can inflict damage upon anything that passes within its radius. A variant on this strategy is the "flail" method, in which the head and single arm are replaced with a linked chain of heads and arms, which may vary widely in size and complexity. More generally, "whipping appendages" were widespread. A different, less common approach is the "steamroll" method, in which a large individual composed of regular segments (each endowed with a powerful propelling appendage) repeatedly bumps into the opponent at full speed, constantly pushing it away in the process.

Among strategies that made use of external sensors, a simple one is the "directed worm" technique, in which a simple crawling worm (a straight chain of aligned limbs, propelling itself through transversal oscillation) is able to consistently move towards its opponent by using sensor input. A variation is the "directed tail", where a complex individual ensures that a swinging tail is constantly directed towards its opponent. Another common occurrence is the two-armed crawler, endowed with two symmetric oscillating arms that serve both for propulsion and attack. By modulating the orientation of arms with sensor input, the creature is able to move towards its opponent.

Besides such identifiable categories, we observed a multitude of idiosyncratic morphologies, ranging from the very simple to the relatively complex. Consider, for example, the larger creature in the top-right picture in Figure 3. The functional portion of its neural network is displayed in Figure 4. Besides the use of mutual connections between the proprioceptors and actuators of various limbs to create synchronised oscillation patterns (and thus efficient locomotion), we see that the "head" contains various connections from external sensors which allow the entire creature to home in on its opponent.

5 Conclusion

We have implemented a system for evolving physical combat among 3D creatures. The system proved consistently successful in evolving competent fighters. We observed a wide range of morphologies and behaviours, ranging from the simple to the relatively complex. The success of this system indicates that physical combat can be used for further experiments involving virtual creatures.

References

1. Sims, K.: Evolving virtual creatures. In: SIGGRAPH 1994, pp. 15–22. ACM Press, New York (1994)
2. Sims, K.: Evolving 3d morphology and behavior by competition. In: Brooks, R., Maes, P. (eds.) Procs 4th Intl Works on Synthesis and Simulation of Living Systems (ALIFE IV), pp. 28–39. MIT Press, Cambridge (1994)
3. Marbach, D., Ijspeert, A.: Co-evolution of configuration and control for homogenous modular robots. In: Groen, F. (ed.) Procs of the Eighth Conference on Intelligent Autonomous Systems (IAS8), pp. 712–719. IOS Press, Amsterdam (2004)
4. Mesot, B.: Self-organisation of locomotion in modular robots: A case study. Master's thesis, EPFL, Lausanne (February (2004)
5. Bongard, J., Zykov, V., Lipson, H.: Resilient Machines Through Continuous Self-Modeling. Science 314(5802), 1118 (2006)
6. Bongard, J.C., Pfeifer, R.: Repeated structure and dissociation of genotypic and phenotypic complexity in artificial ontogeny. In: [21], pp. 829–836.
7. Hornby, G.S., Pollack, J.B.: Body-brain co-evolution using L-systems as a generative encoding. In: [21], pp. 868–875
8. Lipson, H., Pollack, J.: Automatic design and manufacture of artificial lifeforms. Nature 406, 974–978 (2000)

9. Komosinski, M.: The world of framsticks: Simulation, evolution, interaction. In: Heudin, J.-C. (ed.) VW 2000. LNCS (LNAI), vol. 1834, pp. 214–224. Springer, Heidelberg (2000)
10. Taylor, T., Massey, C.: Recent developments in the evolution of morphologies and controllers for physically simulated creatures. Artificial Life 7(1), 77–87 (2001)
11. Miconi, T., Channon, A.: An improved system for artificial creatures evolution. In: Rocha, L., Bedau, M., Floreano, D., Goldstone, R., Vespignani, A., Yaeger, L. (eds.) Procs. 10th Intl. Conf. on Simulation and Synthesis of Living Systems (ALIFE X), MIT Press, Cambridge (2006)
12. Chaumont, N., Egli, R., Adami, C.: Evolving Virtual Creatures and Catapults. Artificial Life 13(2), 139–157 (2007)
13. Shim, Y., Kim, C.: Evolving Physically Simulated Flying Creatures for Efficient Cruising. Artificial Life 12(4), 561–591 (2006)
14. Lassabe, N., Luga, H., Duthen, Y.: A new step for artificial creatures. In: Procs 1st IEEE Conference on Artificial Life (IEEE-ALife 2007), vol. 243, IEEE Press, Los Alamitos (2007)
15. Channon, A.D.: Unbounded evolutionary dynamics in a system of agents that actively process and transform their environment. Genetic Programming and Evolvable Machines 7(3), 253–281 (2006)
16. Hraber, P.T., Jones, T., Forrest, S.: The ecology of Echo. Artificial Life 3(3), 165–190 (1997)
17. Yaeger, L.: Computational genetics, physiology, metabolism, neural systems, learning, vision and behaviour or polyworld: Life in a new context. In: Langton, C.G. (ed.) Artificial Life III, Vol. XVII of SFI Studies in the Sciences of Complexity, pp. 263–298. Addison-Wesley, Reading (1994)
18. O'Kelly, M.J.T., Hsiao, K.: Evolving mutually perceptive creatures for combat. In: Vogt, P. (ed.) Procs. 9th Intl. Conf. on Simulation and Synthesis of Living Systems (ALIFE IX), MIT Press, Cambridge (2004)
19. Miconi, T., Channon, A.: Analysing coevolution among artificial creatures. In: Talbi, E.-G., Liardet, P., Collet, P., Lutton, E., Schoenauer, M. (eds.) EA 2005. LNCS, vol. 3871, Springer, Heidelberg (2006)
20. Cliff, D., Miller, G.F.: Tracking the red queen: Measurements of adaptive progress in co-evolutionary simulations. In: Morán, F., Merelo, J.J., Moreno, A., Chacon, P. (eds.) ECAL 1995. LNCS, vol. 929, pp. 200–218. Springer, Heidelberg (1995)
21. Spector, L., Goodman, E.D., Wu, A., Langdon, W.B. (eds.): Proceedings of the GECCO 2001 conference. Morgan Kaufmann, San Francisco (2001)

Real-Time, Non-intrusive Speech Quality Estimation: A Signal-Based Model

Adil Raja and Colin Flanagan

Department of Electronic and Computer Engineering,
University of Limerick, Limerick, Ireland
{adil.raja,colin.flanagan}@ul.ie
http://www.ul.ie/wireless

Abstract. Speech quality estimation, as perceived by humans, is of vital importance to proper functioning of telecommunications networks. Speech quality can be degraded due to various network related problems. In this paper we present a model for speech quality estimation that is a function of various time and frequency domain features of human speech. We have employed a hybrid optimization approach, by using Genetic Programming (GP) to find a suitable structure for the desired model. In order to optimize the coefficients of the model we have employed a traditional GA and a numerical method known as linear scaling. The proposed model outperforms the ITU-T Recommendation P.563 in terms of prediction accuracy, which is the current non-intrusive speech quality estimation model. The proposed model also has a significantly reduced dimensionality. This may reduce the computational requirements of the model.

Keywords: Non-Intrusive, Signal-based, GP, MOS.

1 Introduction

Speech quality may be reduced due to various reasons in a telecommunications network. Some of these may be the noisy/faulty channels and links, frame loss due to irrecoverable errors and low bitrate coding. Speech quality estimation is vital to the evaluation of quality of service offered by a telecommunications network. Traditionally, speech quality is estimated using subjective tests. In subjective tests, the quality of a speech signal under test is evaluated by a group of human listeners who assign an opinion score on an integral scale ranging between 1 (bad) to 5 (excellent). The average of these scores, termed the *Mean Opinion Score (MOS)*, is considered as the ultimate determinant of the speech quality [1]. Subjective tests are, however, time consuming and expensive. To make up for these limitations, there has been a growing interest in devising software based objective assessment models. There are two kinds of objective assessment models, namely, intrusive and non-intrusive. Intrusive models evaluate the quality of a distorted speech signal in the presence of a corresponding reference signal. The current International Telecommunications Union (ITU-T)

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 37–48, 2008.
© Springer-Verlag Berlin Heidelberg 2008

recommendation P.862 (PESQ) [2] is an example of such an approach. Non-intrusive models, on the other hand, do not enjoy this privilege and base their results solely on the *estimated* features of the signal under test. For this reason, the results of the latter type of models are generally considered inferior to those of the former.

Non-intrusive models can be classified either as *signal-based* or *parametric*. As the name suggests, signal-based models are based on the digital signal processing of human speech. An example of such a model is the current, state-of-the-art, ITU-T Recommendation P.563 for *single-ended* estimation of speech quality [3]. Parametric models, on the other hand, base their results on various properties relevant to the telecommunications network. In the case of Voice over Internet Protocol (VoIP), for instance, these may be transport layer metrics such as packet loss, jitter and end-to-end delay of a call. An example of a parametric model is the ITU-T G.107, commonly referred to as the E-model [4]. Both types of models have their own advantages and limitations. Thus, for instance, signal based models are used to analyse speech quality when the spectral envelope of the speech signal may have suffered from degradation over time. This may happen due to low bitrate coding or transmission over noisy wireless links. Parametric models may be advantageous in VoIP, for instance, where the speech signal may have undergone packet loss, and the speech quality may be estimated as a function of packet loss statistics. A limitation of signal-based models is that they are compute intensive, whereas parametric models are real-time amenable. Moreover, since parametric models are designed for a particular type of communications network, their predictions for that type of network are more accurate than those of signal-based models; signal-based models are suitable for general predictions for a wider variety of networks.

In this paper we propose a new non-intrusive signal based speech quality estimation model based on evolutionary algorithms. In particular we have employed a hybrid optimization approach that uses Genetic Programming (GP) to search for a suitable structure for the desired solution. Coefficients of the models evolved by GP are tuned simultaneously using a Genetic Algorithm (GA) and a numerical method known as *linear scaling*. It is worth mentioning here that that to the best of the authors' knowledge this is the first ever application of evolutionary algorithms for deriving a *signal* based model for non-intrusive speech quality estimation. In the past the authors applied GP along with linear scaling to derive a parametric model as reported in [5]

The main advantage of using GP is that it can produce human-readable results in the form of analytical expressions. Moreover, GP is capable of weeding out irrelevant parameters while concentrating on the most salient ones. These features of GP make our research superior to the past approaches based on various machine learning approaches, as reflected in the results.

The rest of the paper is organized as follows. Section 2 entails a discussion on the nature of signal based models. In section 3 we discuss the speech material used in this research and the various distortion conditions. Section 4 discusses the various experimental details and test results. Section 5 is the conclusion.

2 Signal Based Non-intrusive Models

Signal based non-intrusive models are preferable to parametric ones for various reasons. Firstly, parametric models can be used only with certain types of networks, such as VoIP. Secondly, signal based models are more general in the sense that they are applicable for a wider range of distortion conditions. Unlike the parametric models these models process the audio stream to extract the information relevant to distortions in a signal. The estimated distortions are then converted into MOS for that audio stream. Given this, a signal based model may have two main modules. 1) A feature extractor that processes the speech signal and extracts cogent distortion indicators. 2) A mapping module that transforms the extracted features into MOS estimates. In what follows, some of the well known algorithms that have been used in the past for both feature extraction and MOS mapping are briefly described.

2.1 Feature Extraction Algorithms

Feature extraction algorithms may involve time and/or frequency domain analysis of the speech signal under test. Time domain analysis may involve computation of distortions relevant to the waveform of the speech signal. Some distortions include temporal clipping, level variation and abrupt changes in the temporal envelope of the signal. Frequency domain analysis techniques models normally emulate the human vocal production system [6], or the auditory processing system [7]. ITU-T Recommendation P.563 is the current standard for signal based non-intrusive speech quality estimation. It entails a rigorous feature extraction process that involves the computation of plausible features from both time and frequency domain representations of the signal under test. The overall structure of the P.563 algorithm is divided into three stages. The first is a preprocessing stage in which the signal is level normalized. After this, two additional versions of the distorted signal are created. The first is created by a filter having a frequency response similar to the modified intermediate reference system (IRS) as described in ITU-T P.830 [8]. IRS emulates the frequency response of a standard telephony handset. The second version of the normalized signal is created by using a fourth-order Butterworth high-pass filter with a 100-Hz cutoff frequency and a flat response for higher frequencies, thus emulating the frequency response of cordless and mobile phones. Voice Activity Detection (VAD) is also a part of the preprocessing stage that is used to discard speech sections shorter than 12 ms and to join speech sections separated by less than 200 ms. The second stage pertains to distortion classification which is applied on the preprocessed versions of the signal. Distortion classification is based on three basic principles. The first principle models the human vocal tract as a series of concatenated tubes to reveal the anomalies in the speech signal as a function of abnormal variations in the tubes' sections. The statistics relevant to these anomalies form the speech features.

The human vocal production system may be considered to have three components: lungs as a source of air pressure, vocal chords as source of modulation and

the vocal tract as a resonating source. Thus for *voiced* sounds, the air pressure created by the lungs excites the vocal chords to create a low frequency, quasi periodic sound. The spectral content of this sound is changed due to resonating characteristics of the vocal tract. While speaking, the shape of the vocal tract is changed due to controlled contractions and relaxations of its muscles. This changes the resonant frequencies of the vocal tract, and consequently the spectral content of the speech. To this end, Gray attempted to capture the speech distortions, caused by communications networks, by employing a human vocal production model [6]. The vocal tract is modeled as a set of concatenated tubes with uniform, time-varying cross-sectional areas. Here, it is assumed that most types of speech distortions cannot be produced by a human vocal tract due to the limited and restrained movement of the vocal tract muscles. In general terms, an implausible change in any of the tubes' sizes is considered as a distortion.

The second principle entails a reconstruction of a *pseudo* reference signal from the signal under test to perform an intrusive quality evaluation of the speech signal to estimate the effect of distortions revealed during reconstruction. Signal reconstruction is done by performing a 10th order *linear predictive (LP)* analysis of 5 ms frames of the distorted signal. LP coefficients are converted to *line spectral frequencies* followed by quantization to constrain them to fit the vocal tract model of a typical human talker. LP is a popular speech analysis technique used to represent characteristics of speech with a reduced set of parameters [9][pp280-291]. These quantized coefficients are used to reconstruct the pseudo reference signal. The difference between the pseudo reference signal, in a spectral sense, and the signal under test gives a basic quality estimate that is used as a feature for overall quality estimation.

The third principle is to determine specific distortions encountered in voice channels, such as temporal clipping, frame erasures, signal correlated and background noise, robotization and level variation etc.

According to the reference implementation of the algorithm, a total of 43 features are extracted that depict various characteristics of the speech signal under test. All features are divided into various distortion classes. Based on a restricted set of *key parameters*, an assignment to a dominant distortion classes is made. A complete description of these features is skipped here for brevity, but they can be into three distortion groups pertaining to: 1) Unnaturalness of speech, 2) noise, and 3) interruptions, mutes and temporal clipping.

2.2 Mapping Algorithms

Once cogent features corresponding to the speech signal under test have been extracted, they are mapped to the speech quality using an appropriate regression tool or a machine learning algorithm. Thus, for training a model numerous MOS-labelled speech databases are used. An MOS-labelled speech database may have a considerable number of speech samples from both male and female speakers, and possibly in different languages. The duration of each speech sample may vary from 8-12 secs. Each speech sample may be affected by a certain type of network distortion, such as frame erasure, bit errors and/or signal correlated/

uncorrelated noise. Each sample also has a MOS score associated with it, derived normally from subjective tests [1]. The features relevant to distortions for all the samples serve as the input domain variables and the corresponding MOS scores form the target values for learning. After learning completes, the derived model is also tested and validated using unseen data from a separate set of speech samples/databases, as a standard practice.

Numerous learning algorithms have been used in the past to map the effect of speech features, and/or their relevant statistics, to speech quality. Depending upon the learning algorithm the training and mapping procedures may vary. One approach is to compute a significant number of feature vectors corresponding to clean, distortion free, speech files. A database of clean speech feature vectors may be formed by classifying the latter into clusters to form a reference code-book. An appropriate vector quantization algorithm such as K-means, as in [10], or self organizing maps, as in [11], may be employed. As a part of training, feature vectors corresponding to distorted speech samples are extracted and their distances are computed from the best matching feature vector in the reference code-book in a Euclidean sense. These auditory distances are eventually mapped to reference MOS scores using a 2nd or a 3rd order polynomial. An obvious limitation of such an approach is the time required to search for a best matching vector from the reference code-book of feature vectors of clean speech. Another approach is to map the feature vectors of the training speech samples directly to speech quality using an artificial neural network [12]. In [13] Falk and Chan have used Gaussian mixture models (GMMs), support vector classifiers and multi adaptive regression splines at various stages of their proposed algorithm to map the cogent features to speech quality. Similarly in [14] Grancharov et al. also employed a GMM for speech quality prediction. In [15] Li and Kubichek employed a hidden Markov model (HMM) for mapping the speech related features to quality. Among all of these algorithms HMMs attempt to explore statistical dependencies between adjacent segments of human speech, whereas for the rest of the algorithms aggregated values of features over the entire length of speech signal are used.

ITU-T P.563 uses a two step mapping process. First, an initial quality estimate is made that is a linear combination of the values of a subset of speech features that fall under a particular distortion class. Second, a final quality estimate is made that is again a linear combination of the initial quality estimate and 11 additional features. P.563 has shown a high correlation with the human evaluation of speech quality, ranging between 0.88–0.90 [3] for various ITU-T benchmark tests.

2.3 Proposed Model

In this paper we have proposed a new model for speech quality estimation. We have used P.563 as the feature extraction algorithm in this research. This has been chosen for two reasons: 1) P.563 is the current, state-of-the-art standard for non-intrusive speech quality estimation. 2) it computes the most numerous and

most varied features relevant to speech quality than any other feature extraction algorithm. However, for mapping the features to speech quality we have employed a GP based symbolic regression approach, along with a traditional GA, and linear scaling as proposed by Keijzer in [16], for parameter optimization. GP is used to evolve a suitable structure for mapping the features to speech quality. GP is also known to prune off the redundant features and to retain the most useful ones in the genome of the final individual. The GA is employed to fine tune the numeric leaf values during evolution.

3 Speech Databases

A total of eight MOS labeled speech databases were used in this research. Out of these, seven *multilingual* databases belong to the ITU-T P-series supplement 23 (Experiments 1 and 3) [17]. These databases include 1328 speech samples distorted due to conditions such as *signal correlated noise, transcoding, bit errors and frame erasures*. The databases include utterances by male and female speakers. The eighth database includes 240 utterances in North American English accent by two male and two female speakers with seven types of distortion conditions. This database is compiled by Nortel Networks [18]. The distortion conditions, each of varying levels, include *signal correlated noise, coding distortions, tandeming, temporal clipping, bit errors and speech level variation*. 70% of the speech files, and their corresponding MOS, in these databases were dedicated for training and the remaining 30% for testing reasons. More specifically, input/output patterns of 1,100 speech files were picked randomly as training data, and the remaining, 468 patterns were used for the purpose of testing.

It is worth describing here the meaning of various distortion conditions mentioned above. Signal correlated noise (also known as multiplicative noise or modulated noise) is a function of the amplitude of the speech signal. It is introduced by *waveform* codecs due to quantization of the amplitude. Some examples are logarithmically companded PCM (ITU-T G.711) [19] and ADPCM (ITU-T G.726) [20]. Transcoding (or codec tandeming) occurs when the speech signal is processed by more than one codec in the transmission path. This happens in scenarios where participants of a call use different codecs to communicate with each other. In a digital transmission channel the speech signal or its coded version may undergo bit errors, as in wireless networks. A frame erasure occurs when a coded speech frame undergoes an irredeemable error, as in wireless networks, or when a frame is lost entirely, as in an event of a packet loss in VoIP. Codec distortions correspond to the degradations induced by the underlying speech coding/compression scheme employed on the transmission channel. Temporal clipping occurs when a speech codec employs a voice activity detection (VAD) algorithm for silence suppression. In this, silence intervals during speech are captured and suppressed from being transmitted to the receiver to achieve bandwidth saving. VAD results in front-end clipping during the start of a speech segment and may lead to an audible distortion. Finally, level variation corresponds to abrupt changes in the volume of speech.

4 Experiments and Results

4.1 Experimental Setup

As a first step feature extraction was performed by processing the MOS labeled speech databases discussed in section 3 using the P.563 algorithm. Values of 43 features corresponding to each of the speech files were accumulated as the input domain variables. The corresponding MOS scores formed the target values for training and testing the evolutionary experiments.

Two GP experiments were conducted. The common parameters of both experiments are listed in Table 1. In both of these population size was set to 3,000. Each experiment was composed of 50 runs, each spanning 100 generations. Tournament selection with Lexicographic Parsimony Pressure (LPP) [21] was used in both experiments. An initial maximum tree depth of 6 was used. The maximum tree depth was changed dynamically with an upper limit of 17. Survival was based on elitism. The elitist criterion was such that at each generation the depth of the best individual would be noted. Any individuals in the child population exceeding this depth would be removed from evolution at this stage as a first step. Next, up to half of the entries of the new population would be filled up with the remaining individuals from the offspring population on the basis of fitness. The remaining slots in the new population would be filled with the most fit individuals from the parent population.

Table 1. Common Parameters of GP experiments

Parameter	**Value**
Population Size	3,000
Initial Tree Depth	6
Selection	LPP Tournament
Tournament Size	7
Genetic Operators	Crossover, Subtree Mutation and Point Mutation
Operator Probabilities	0.95, 0.1, 0.1
Survival	Elitist
Function Set	+, -, *, /, sin, cos, log_{10}, log_e, sqrt, power.
Terminal Set	Random numbers [-6–6]. P.563 features.

In both of the experiments scaled mean squared error (MSE_s) was used as the fitness criterion and is given by equation (1).

$$MSE_s(y,t) = 1/n \sum_{i}^{n} (t_i - (a + by_i))^2 \tag{1}$$

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 the corresponding MOS. a and b adjust the slope and y-intercept

of the evolved expression to minimize the squared error. They are computed according to equation (2).

$$a = \overline{t} - b\overline{y}, b = \frac{cov(t,y)}{var(y)} \tag{2}$$

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 [16]. Instead of using *protected* functions, 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 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. [22]. 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.

The first experiment (referred to as experiment 1) was based purely on GP. In the second experiment (referred to as experiment 2) the leaf coefficients of the GP trees were tuned using a GA during evolution. Various meta-heuristic algorithms and numerical methods have been employed by researchers in the past for tuning the leaf coefficients to enhance the fitness of GP trees. For instance, in [23] Howard and Donna proposed a hybrid GA-P algorithm, where GP was used to find optimal expressions for problem solving and a GA was used to tune the coefficients of the GP trees/expressions during evolution. Similarly in [24] Topchy and Punch have used the gradient descent algorithm for the local search of leaf coefficients of GP trees. Moreover, quasi-Newton method has been used to achieve the same objective in [25]. As a tradeoff between fitness enhancement and computational efficiency, our implementation of the evolutionary algorithm employed a GA to fine tune the coefficients of 30 best GP trees of every generation was performed. The coefficients learnt by the GA based tuning were coded back in to the respective GP trees. It was hoped that this would enhance the overall fitness of the subsequent populations as the genetic material of these possibly more fit GP trees would propagate to the subsequent generations. Here a simple GA was implemented with genes of type *double*. A population of size 100 was used with 15 generations per run. Single point crossover and mutation were used as the the genetic operators with probabilities equal to 0.8 and 0.2 respectively.

4.2 Results and Analysis

Table 2 lists the statistics about the MSE_s of the training/testing data and final tree size (in terms of number of nodes) of the best individuals of the two experiments under consideration. The fitness statistics relevant to experiment 2 are generally better as compared to experiment 1 over both training and testing

data. Nonetheless, a Mann-Whitney-Wilcoxon test was also performed to formally decide if a significant difference exists between the simulations at a 5% significance level. The significance test did not reveal any difference between the two experiments, and consequently between the two approaches. However, the best individual, in terms of minimum MSE_s over the testing data, belongs to experiment 2, as can be seen in Table 2.

Table 2. Statistical analysis of the GP experiments and derived models

	$Experiment1$			$Experiment2$		
Stats	MSE_{tr}	MSE_{te}	$Size$	MSE_{tr}	MSE_{te}	$Size$
Mean	0.3673	0.3488	35.58	0.3618	0.3441	36.16
Std. Dev.	0.0172	0.0183	13.9972	0.0159	0.0169	17.5875
Max.	0.4049	0.4026	70	0.3885	0.3817	102
Min.	0.3239	0.3146	12	0.3271	0.3071	18

Performance results of the best individual over the testing data are shown in Table 3 and a comparison with the reference implementation of ITU-T P.563 is also shown. Table 3 also lists the percentage of Prediction Gain (PG) given by equation (3). This individual is the proposed model and is derived from experiment 2.

$$\%PG = \frac{MSE_{P.563} - MSE_p}{MSE_{P.563}} \times 100 \tag{3}$$

where $MSE_{P.563}$ and MSE_p represent the MSE of ITU-T P.563 and the proposed model with respect to reference MOS respectively.

Table 3. Performance results of the proposed model versus the reference implementation of ITU-T P.563 in terms of MSE_s

	ITU-T P.563	GP Based Model	Percentage Enhancement
Training	0.3937	0.3415	9.89
Testing	0.3674	0.3071	16.41

The proposed model has 85 nodes (including terminals and functions), however, it is a function of only *9* features as opposed to *43* features of the reference implementation of ITU-T P.563. This may prove beneficial in reducing the computational requirements of the algorithm. The model is not given here due to shortage of space, however, the independent variables (i.e. features of P.563) are briefly discussed as follows:

– *Average pitch:* This feature is used as a basic speech descriptor. An autocorrelation method is used to compute pitch period estimates of 64 ms *voiced* frames. Average pitch is one of the distortion classifiers and is used mainly to differentiate between unnatural male and female voices. It is also used by ITU-T P.563 to formulate the initial estimate of quality.

- *Final VTP average:* VTP refers to an array that stores the cross sectional areas of the emulated vocal tract tubes, as described in the first principle of ITU-T P.563 in section 2.1. Final VTP average relates to the mean of the area of last tube over the entire length of the signal.
- *ART average:* ART (articulators) are formed by aggregating the elements of the VTP elements into three groups. These groups correspond to the front, middle and rear *cavities* of the human vocal tract. This feature represents the average size of the rear cavity.
- *Basic voice quality:* This feature is derived from the second principle of ITU-T P.563 described in section 2.1.
- *LPC kurtosis, LPC skewness and absolute LPC skewness:* These three features represent statistics relevant to the 21 (LPC) *linear predictive coefficients* of the speech signal. The statistics are computed for the LPCs of each frame and aggregated over all frames of the signal. Skewness and kurtosis are the 3rd and 4th moments about the mean and are considered to give meaningful insights into the spectral characteristics of the signal.
- *Spectral clarity:* This feature is computed for voiced sections of the speech signal to be analyzed. It corresponds to the difference between the values of harmonics of pitch and the non-harmonic spectral components in the gaps between the harmonics. First five harmonics of the pitch are used. FFT is used for spectral estimation.
- *Estimated segmental SNR:* This feature is used to detect the presence of signal correlated noise.

5 Conclusion

In this paper we have presented a novel signal based method for non-intrusive evaluation of speech quality. We employed the ITU-T P.563 algorithm for speech feature extraction. Estimates of speech quality (MOS) from subjective tests have been used as (reference) target values. Mapping between the various features and the reference speech quality is obtained by GP based symbolic regression. Two GP experiments were performed. The first was purely based on GP, with *scaled* MSE as the fitness function. The second experiment additionally employed a hybrid approach in which the coefficients of selected individuals were tuned using a GA, during every generation of GP based evolution. Both experiments have produced individuals that outperform the reference implementation of ITU-T P.563. Although it was expected that the hybrid optimization approach would produce better individuals, the obtained results were not significantly different from the first experiment. However, the best individual was produced by the second experiment.

The best individual, in terms of fitness over testing data, has been proposed as a model for quality estimation. This model, being a function of only 9 features, as opposed to 43 features of ITU-T P.563's reference implementation, is one of reduced dimensionality too. This is also a significant result of this research. Our future goal is to benchmark the proposed model to investigate if any computational performance gains can be achieved.

References

1. ITU-T.: Methods for subjective determination of transmission quality. International Telecommunications Union, Geneva, Switzerland, ITU-T Recommendation P.800 (1996)
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, ITU-T Recommendation P.862 (2001)
3. ITU-T.: Single-ended method for objective speech quality assessment in narrowband telephony applications. International Telecommunications Union, Geneva, Switzerland, ITU-T Recommendation P.563 (2005)
4. ITU-T.: The E-Model, a computational model for use in transmission planning. International Telecommunications Union, Geneva, Switzerland, ITU-T Recommendation G.107 (2005)
5. Raja, A., Azad, R.M.A., Flanagan, C., Ryan, C.: Real-time, non-intrusive evaluation of VoIP. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 217–228. Springer, Heidelberg (2007)
6. Gray, P., Hollier, M.P., Massara, R.E.: Non-intrusive speech-quality assessment using vocal-tract models. In: IEE Proceedings of Vision, Image and Signal Processing, vol. 147 (December 2000)
7. Hermansky, H.: Perceptual Linear Predictive (PLP) Analysis of Speech. Journal of Acoustical Society of America 87(4), 1738–1752 (1990)
8. ITU-T.: Subjective performance assessment of telephone-band and wideband digital codecs. International Telecommunications Union, Geneva, Switzerland, ITU-T Recommendation P.830 (1996)
9. Gold, B., Morgan, N.: Speech and Audio Signal Processing: Processing and Perception of Speech and Music. Wiley, New York (1999)
10. Jin, C., Kubichek, R.: Vector quantization techniques for output-based objective speechquality. In: IEEE International Conference on Acoustic, Speech and Signal Processing (ICASSP), vol. 1, pp. 1291–1294 (November 1996)
11. Picovici, D., Mahdi, A.E.: New output-based perceptual measure for predicting subjective quality of speech. In: IEEE International Conference on Acoustic, Speech and Signal Processing (ICASSP), vol. 5, pp. 17–21 (May 2004)
12. Tarraf, A., Meyers, M.: Neural network-based voice quality measurement technique. In: IEEE international symposium on Computers and Communications, pp. 375–381 (July 1999)
13. Falk, T.H., Chan, W.-Y.: Single-ended speech quality measurement using machine learning methods. IEEE Transactions on Audio, Speech and Language Processing 14(6), 1935–1947 (2006)
14. Grancharov, V., Zhao, D.Y., Lindblom, J., Kleijn, W.B.: Low-complexity, nonintrusive speech quality assessment. IEEE Transactions on Audio, Speech and Language Processing 14(6), 1948–1956 (2006)
15. Li, W., Kubichek, R.: Output-based objective speech quality measurement using continuous Hidden Markov Models. In: Seventh International Symposium on Signal Processing and Its Applications, vol. 1, pp. 1–4 (July 2003)
16. Keijzer, M.: Scaled symbolic regression. Genetic Programming and Evolvable Machines 5(3), 259–269 (2004)

17. ITU-T.: coded-speech database. International Telecommunications Union, Geneva, Switzerland, ITU-T P.Supplement 23 (1998)
18. Thorpe, L., Yang, W.: Performance of current perceptual objective speech quality measures. In: IEEE International Speech Coding, vol. 1, pp. 144–146 (May 1996)
19. ITU-T.: Pulse Code Modulation (PCM) of voice frequencies. International Telecommunications Union, Geneva, Switzerland, ITU-T Recommendation G.711 (November 1988)
20. ITU-T.: 40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM). International Telecommunications Union, Geneva, Switzerland, ITU-T Recommendation G.726 (1990)
21. Luke, S., Panait, L.: Lexicographic parsimony pressure. In: W.B.L. (ed.) GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, pp. 829–836 (2002)
22. Gustafson, S., Burke, E.K., Krasnogor, N.: On improving genetic programming for symbolic regression. In: D.C., et al. (eds.) Proceedings of the 2005 IEEE Congress on Evolutionary Computation, Edinburgh, UK, 2-5 September, vol. 1, pp. 912–919. IEEE Press, Los Alamitos (2005)
23. Howard, L.M., D'Angelo, D.J.: The GA–P: A genetic algorithm and genetic programming hybrid. IEEE Expert 10(3), 11–15 (1995)
24. Topchy, A., Punch, W.F.: Faster genetic programming based on local gradient search of numeric leaf values. In: Spector, L., Goodman, E.D., Wu, A., Langdon, W.B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M.H., Burke, E. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), San Francisco, California, USA, july 7-11, pp. 155–162. Morgan Kaufmann, San Francisco (2001), http://www.cs.bham.ac.uk/~wbl/biblio/gecco2001/d01.pdf
25. Mugambi, E.M., Hunter, A., Oatley, G., Kennedy, L.: Polynomial-fuzzy decision tree structures for classifying medical data. Knowledge-Based Systems 17(2-4), 81–87 (2004)

Good News: Using News Feeds with Genetic Programming to Predict Stock Prices

Fiacc Larkin and Conor Ryan

Biocomputing and Developmental Systems
Computer Science and Information Systems
University of Limerick, Ireland
{fiacc.larkin,conor.ryan}@ul.ie
http://bds.ul.ie

Abstract. This paper introduces a new data set for use in the financial prediction domain, that of quantified News *Sentiment*. This data is automatically generated in real time from the Dow Jones network with news stories being classified as either Positive, Negative or Neutral in relation to a particular market or sector of interest.

We show that with careful consideration to fitness function and data representation, GP can be used effectively to find non-linear solutions for predicting large intraday price jumps on the S&P 500 up to an hour before they occur. The results show that GP was successfully able to predict stock price movement using these news *alone*, that is, without access to even current market price.

1 Introduction

Stock market price prediction has long been an attractive area for research, with techniques including everything from Neural Networks [1][3][2][4] to Genetic Programming [5][6] being used to try and predict stock price movement. These methods typically base their predictions on factors such as recent prices in the market. This is despite the Efficient Market Hypothesis (EMH) [11], which states that financial markets are "informationally efficient", that is, stock prices immediately reflect all known pertinent information so that it is not possible to outperform the market using information which is *already known to the market*.

While the EMH would write off any success to luck, effectively saying that one is as likely to have the same success rolling chicken bones as running GP, these predictive methods gamble on having access to *high quality* information that no one else has. In particular, although the same raw information (typically stock prices) is available to everyone, not everyone has the ability to analyse it in useful ways, and so, there is opportunity to profit while the market adjusts its prices, as an *unused* source of information may give investors an advantage.

This paper considers a different source of information, *news stories*. Although the basic idea that there is a relationship between news events and stock market price movements is not a new one [7] there has been very little work done to

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 49–60, 2008.
© Springer-Verlag Berlin Heidelberg 2008

incorporate news events into quantitative style models. This may be due to the fact that the human interpretive element of news stories does not easily lend itself to the quantitative scrutiny that is regularly applied to the so called *hard* data such as employment numbers or interest rates.

However, if one could employ news stories/events in a quantitative and automatic way, then this could give one an enormous advantage in the market, in the sense that it would be possible to react more quickly than the market.

Recently, a research company (RavenPack International, S.L.) has developed means for quantifying news stories; and our goal was to search for a model based on news sentiment (i.e. whether a news story is relevant to the particular market or sector, and if it is positive, neutral or negative) that exhibited predictive behaviour for intraday price movements. We wish to find if there is a relatively straightforward way of combining only inputs from news sentiment to give predictions on future intraday price movement.

We start with a description of the kind of data that we are dealing with and the sort of data pre-processing we performed. Next, in section 4, we build up an experimental approach that uses *only* news stories as inputs, before demonstrating in sections 5 and 6 that we can successfully predict stock price movement in the S&P 500 index statistically significantly better than a standard benchmark approach.

2 Background

Understanding how to utilise news is a difficult task even when in a pre-quantified form. The data are noisy, containing strong oscillating cycles based around cultural work practises. Two distinct functions are at play within the same data; speculative and reflective reporting, those that happen before and after events respectively and they have very different characteristics from each other.

Despite the huge search space that has to be covered to find equations for the complex interactions at play, this problem has only a relatively small number of inputs, those being the raw news stories released at each time point.

Just as complicated metrics can be derived from simple price, such as the beta coefficient used in the financial analysis of a company compared to a sector or portfolio, so too should it be possible to construct more descriptive terms from raw news inputs. Evidence that a small number of simple inputs can manifest into non-linear behaviours indicative of financial time series' was demonstrated by Lawrenz & Westerhoff [10] who constructed an artificial exchange traded on by a few Genetic Algorithms who learnt to dynamically adjust the coefficients of basic technical analysis strategies in relation to the stochastic influx of news flows and the reactions of the other trading agents. From this basic system a time series was produced that exhibited a number of the unusual artifacts well documented in the financial literature [7] such as the tendency for the extremities of the distributions of price return to be more dense than a Gaussian curve would explain and the clustering of volatility and trading volume.

3 The Data

Stories published electronically over the Dow Jones network are classified by RavenPack as being either Positive, Negative or Neutral for a particular market or sector. No other information is given, nor are stories within a particular class ranked, i.e. a story is either in a class or no. It is necessary to make the contextual distinction as the same story can have very different interpretations from different points of view - consider the difference in effect that reports of political turmoil in a region would have on the price of oil vs an indigenous technology company.

The classification procedure is done with a propriety technology in real time, the details of which are beyond the scope of this paper. Upon initially inspecting the three time series a number of things become very evident;

1. All three series of news stories seem to maintain a stable ratio with a tight variation between them even though the total numbers of news stories fluctuates from year to year and week to week.
2. There is always a bias toward more positive stories than negative ones. The well known slight upward bias [13] observed in the markets over time would not appear to explain the magnitude of the difference between positive and negative stories which usually averages at around two to one.
3. Viewed from different time scales news has strong periodic tendencies, the most notable of which have cycles spiking every 91 days; with the quarterly earnings season (Fig 1), 7 days; bulging at midweek with virtually no news at the weekends and intraday spikes occurring at around 7:30 and 16:30 daily.
4. Visual inspection (Figure 2) dispels any naive notion that a *simple* correlation exists between current news sentiment and market movement.
   Pearson's correlation tests done between the number of positive, neutral, negative news stories vs the S&P itself, show the absence of a straight linear correlation (Table 1). It is simply not the case to expect that increases in positive news stories will instantly be reflected in market movement, any present relationship is far more complicated than that.

Table 1. The Pearson's correlations between three news sets and the S&P 500 Index

Correlation	Story Sets
-0.0058	All Stories
-0.0046	Positive Stories
-0.0177	Negative Stories

3.1 Pre-processing

Like all experiments dealing with financial data, a number of pre-processing steps had to be applied to the data with great care given to avoid inadvertently contaminating prior data points with future information. We used data from January to September of 2007 at one minute's resolution. The Standard & Poor's

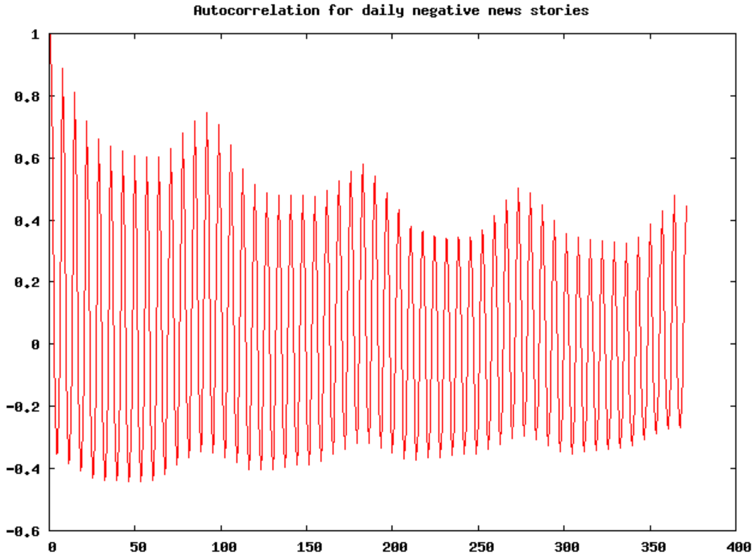


Fig. 1. Auto correlation with lags from 1 to 365 days. The time series is the daily totals of negative news story. note the quarterly peaks every 91 days.

500 index was chosen as our target market. The above mentioned periodicity must be removed for each of the time scales, the financial literature contains many ways to achieve this with exotic filters [9] most of which suffer from being black boxes where it is hard to verify if future information has been brought backwards in time. Fortunately, the very simple technique of subtracting the value x points behind (where x is the cycle period) works effectively. With this in mind we make three passes over the data one removing the 91 day cycle, one removing the 7 day cycle and one removing the 24 hour cycle. Only after this is done to the 24/7 data series, can all data points representing times outside of the New York trading hours be removed.

To do this we merge the news series with the S&P data which is also necessary to ensure date and time congruence, any points that do not match up are dropped. Points lying significantly outside a series' passed observed range are removed using Data Clipping and the values must then be normalised in some way to make them amenable to the GP operators. The average number of stories per minute conveniently brings the values to manageable levels avoiding the need for a rolling window normalisation. For price itself some form of differencing must be done to detrend the data. Getting the log difference can then help curtail the extremes but we are particularly interested in such movements and so we avoid doing this. Great care must be taken to remove the trans-day data as not to confuse the model into believing for example, that the first minute of Monday morning comes immediately after the closing bell on Friday evening; this would subject the model to sudden price shocks that don't actually exist.

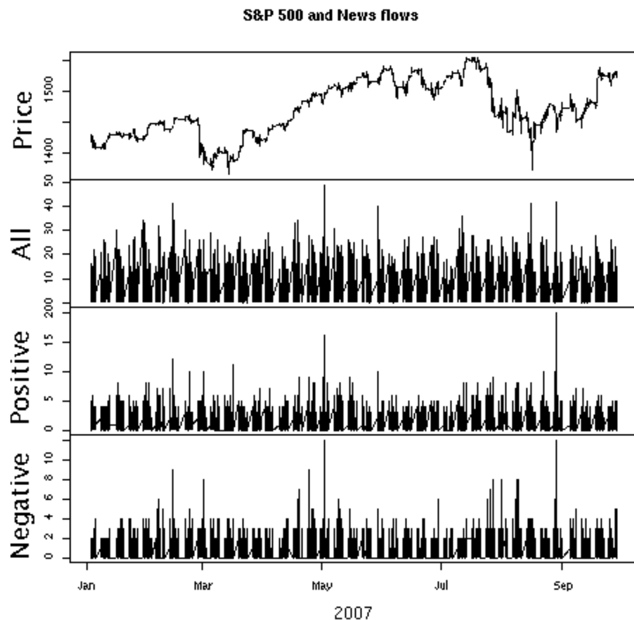


Fig. 2. S&P 500 index along side the raw numbers of All,Positive and Negative stories that should effect it

4 Experiments

Initially we attempted a GP hits based symbolic regression approach to predict raw Δ price movement 20 minutes ahead of the S&P 500 index from a large collection of news inputs. RavenPack's sentiment series (PEQ, BMQ) were used, these are created with proprietary phrase-list and Bayesian algorithms respectively. Both have been designed to classify the effect individual stories will have on the US domestic stock market. Dow Jones supply tags with each story giving information on the topics the story touches upon, these tags are assigned manually by the stories author. Only stories with a specific combination of tags deemed relevant to the US Domestic Market are fed to the classifiers to compile the PEQ and BMQ series. The specific list of relevant filtering tags along with the training examples for the classifiers were arrived at by a group of domain experts. From each series we made six inputs: the three raw **Pos, Neu, Neg** counts, **All** the sum of all stories and two manually constructed series, ξ and ψ which are created using the cumulative daily sums.

$$\xi_t = \frac{\sum\limits_{daily} P_t - \sum\limits_{daily} N_t}{\sum\limits_{daily} P_{t-1} - \sum\limits_{daily} N_{t-1}}$$

$$\psi_t = \frac{\sum_{daily} A_t - \sum_{daily} U_t}{\sum_{daily} A_{t-1} - \sum_{daily} U_{t-1}}$$

Where (P,N,A,U) = (Positive,Negative,All,Neutral) number of stories respectively and the sums are daily running accumulators that reset at the start of each day.

The original 8 RavenPack inputs All, Pos, Neu, Neg for BMQ and PEQ were subjected to the preprocessing regime detailed above, and then the additional two columns of ξ and ψ were added for each making 12 columns of data. These 12 columns taken at time (T), (T-30)minutes and (T-60)minutes make up an input vector of 36 columns. The desired output is the 20 minute Δ price column 20 minutes in the future. This all gives a matrix totalling 37 columns in which every row represents a single GP fitness case with three sets of past news inputs and one future price movement target.

To compensate for the large search space created from the 36 input repertoire GP had to choose from, we ran numerous experiments with large populations of 5000 for 51 generations. The standard way to test the performance of a system such as this is to compare it with a *trivial speculation* method, that is, one that simply guesses the values. If the evolved system is not statistically significantly (a P-value < 0.05) then the solution is of low quality.

These initial experiments did not yield solutions that were significantly better, and analysis showed that there were simply too many inputs, with little improvement being shown as the populations evolved.

4.1 Revised Experiments

While one possible remedy to the situation above is to simply increase the population size and resources being thrown at the problem, we instead designed a second round of experiments that were more in keeping with the original question, which was *is there a predictive relationship between news and price?* rather than *can we predict exact price over fixed time frames?* A number of changes were made to the set up:

First, rather than predicting raw price Δ at a fixed point away we switched the target series to the maximum value Δ from the subsequent hour for every point. Second, we switched from using single point news values at different time lags to using average values per minute over the last 20 minutes, 60 minutes and 1 week. The idea here is to use the general news flow levels over different time periods rather than specific impulses at exact times. With a wider net to sense changes for each data point we hope to leave some leeway in how long it takes traders to react to the release of daily information. The third change in keeping with (Tetlocks [14]) findings on extreme value news involvement was to change to a classifier fitness function with a simpler task of predicting if at any point in the next hour the price would move up beyond two standard deviations of the mean, both of which are calculated from the previous week so as not to use future knowledge. Two standard deviations were chosen as this represents a significant jump in price.

There are over sixty thousand data points of S&P 500 minute values in the data set with only around three thousand positive cases where the subsequent hour held a price move greater then two standard deviations of the past week. This large bias of negative to positive cases would be spotted by GP easily and undoubtedly result in premature solutions that only ever guessed negative.

The training, testing and validating sets were constructed as to give each a representative ratio of the markets positive to negative distribution of cases. This left us with three data sets all of size 13627, all containing 613 positive cases and the rest negative. A modified fitness function was then created to discourage costly false positives but also avoid overly conservative solutions.

$$Standardised\ Fitness = \frac{(\alpha - (\beta + \gamma))}{(\beta + 1)}$$

Where: α=total number of fitness cases, β=number of true positives, γ= number of true negatives and Standardised Fitness is the term we wish to minimise.

False positives represent *long* market positions (buy orders) that fail to increase by the specified amount and may cause losses especially with trading costs considered. We wish to avoid these false positives above all else. However any attempt to add a punishment term to the fitness function that would exacerbate the effects of false positives resulted in overly conservative models that would always bet negatively for fear of getting a positive wrong. Experience showed that better results came from rewarding a combination of both correct positive classifications and overall correctness but with an exaggerated bias toward the correct positives. There are far more negative cases, and so such a function serves as encouragement enough to err on the side of caution without the need for punishment terms. It was also necessary to include a condition that gave solutions with zero correct positives the worst (maximum standardised fitness) score. Such solutions would invariable be in the initial population and erroneously appear to score well under our fitness criteria, ruining the run with premature convergence.

The input set was cut to a total of nine time series, the Positive, Negative and All average news stories per minutes at each of the above mentioned look back periods (20 minutes, 60 minutes and one week). 31 runs of 500 individuals were used for 51 generations in a steady state GP algorithm with tournament selection and ramped half and half initialisation. Two ephemeral random constants {**both greater than 0 but less than 1**} and three statics constants {**2.0, 0.5, 0.01**} where also included as terminals. The function set was made of the basic arithmetic {**+, -, *, p/**}, three modified logic operator {**nAND, nOR, nNOT**} which operate on real floating point values and finally three conditional structures {**IF-less-than-half, nGT, nLT** }.

```
(defmacro IF-less-than-half
    (first-argument then-argument else-argument)
  `(if (< (eval ',first-argument) 0.5)
      (eval ,then-argument)
      (eval ,else-argument)))
```

```
(defun nGT (a b)
  (if (>= a b) 1.0 0.0))

(defun nLT (a b)
  (if (< a b) 1.0 0.0))

(defun nAND (a b)
  (if (and (>= a 0.5) (>= b 0.5)) 1.0 0.0))

(defun nOR (a b)
  (if (or (>= a 0.5) (>= b 0.5)) 1.0 0.0))

(defun nNOT (a)
  (- 1.0 a))
```

5 Results

The best solution was found in one of the runs at generation 38. It had 136 nodes with a depth of 17 and utilises all of the news inputs available. Even though there are only two prediction classes, positive and negative, a base case comparison using a simple coin toss would be very unfair as this would inevitably just reflect that the vast majority of the set contains negative cases. For this reason we use a base case comparison model that makes a random prediction base on a probability distribution bootstrapped from the data, that is to say only one in every 22.23 times is the base case likely to predict a positive. Table 2 compares the best evolved solution against this base case model.

The overall hits (correct predictions; positive or negative) are higher with the distribution aware random predictor but this is not a concern for us as 95% of the set comprises of negatives and so a conservative model will attain most of

Table 2. Comparison of result between the GP found solution and the distribution aware random predictor. Hits are out of 13627 while True Positives are out of 613 leaving 13014 negative cases.

GP Found Solution					
	TP/FP ratio	Standardised Fitness	Hits	TP	FP
Training	3.54	6.38	12612	158	560
Testing	3.14	5.39	12614	187	587
Validation	3.60	6.39	12598	160	576
Comparison Solution					
	TP/FP ratio	Standardised Fitness	Hits	TP	FP
Training	25.21	47.76	12433	24	605
Testing	22.71	45.36	12493	24	545
Validation	24.46	47.04	12451	24	587

these hits. We wish the ratio between the true positives and false positives to be as low as possible. This number serves as the trust (or lack of it) we would have for a positive prediction made by our model. One divided by this number would be the probability value we would use for a Kelly [12] style bet.

The validating data for our model were found to be statistically significant at the (P < 0.001) level showing that news certainly does have predictive power over intraday price movement although such movement as was predicted by our model only represents about 1.1% of the cumulative movement which occurred though out this period.

The base case achieves one correctly predicted positive in every 24.46 positive prediction's made. Our GP solution gets one in every 3.6 positive predictions correct. Such a score may not sound like a lot but one must considers that a correct positive classification means the markets upward movement within the next hour will go beyond a very large threshold whereas the 2.6 remaining incorrect predictions simply mean that this huge jump doesn't occur, but not necessarily that the market will fall in value. One could still end up making money on an incorrect positive classification.

Using the two standard deviations of the previous rolling week could have an unanticipated effect on the model as markets in a phase of gradual volatility change move the expectation of what the target price jumps should be. This may result in more conservative predicting during higher volatility and a more progressive style in smoother times. We did not notice any great effect of this kind. Using a static value would ensure this did not happen although it would be wise to revise this number periodically to keep up with market conditions.

Figure 3 shows the distribution of times when the model gets true positive and false positives. It is interesting to note the tendency for correct predictions to occur in the last hour of trading while the false positives have their median toward the centre. This is unsurprising when we consider the times of the day when the event we are trying to predict actually occur. Even though it is generally known that more volatility occurs just after market open, It would appear the specific behaviour we are looking for (upward movement beyond two standard deviation within an hour) are more likely to happen towards the end of the day. GP without being given any direct time input is not able to pick up on this.

The best GP solution described here was structurally very complex, and unreadable as were all solutions in this run beyond generation 10. Early on in another run when a tree depth restriction of 5 was used an interesting parsimonious solution was found. This simple little solution takes the number of negative divided by positive Average Stories Per Minute (ASPM) over the last hour and multiplies by the positive ASPM over the last week, if the result is greater or equal to 0.5 it will predict a market jump. The performance of this strategy falls between that of our complex GP solution discussed above and the base case predictor. The simplicity of this model affords us the opportunity to analyse why such a model should do any better than average.

$$Primitive\ Solution_i = ((\alpha_i * \frac{\beta_i}{\gamma_i}) >= 0.5)$$

where α= Positive ASPM over the last **week**, β= Negative ASPM over the last **hour** and γ= Positive ASPM over the last **hour**.

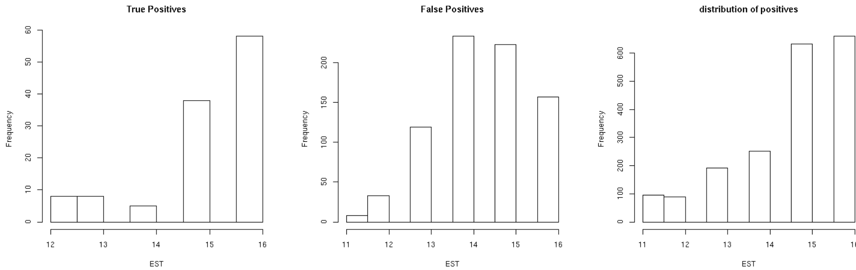


Fig. 3. The times of the day when the system gets True Positives (left), false positives (centre) and the times when the jumps we are trying to predict actually occur (right)

Table 3. Results for the primitive solution. Hits are out of 13854, True Positives are out of 640, leaving 13214 negative cases.

Data	TP/FP ratio	Standardised Fitness	Hits	TP	FP
Training	8.23	14.94	12628	81	639
Testing	9.02	15.46	12540	84	758
Validating	8.21	14.10	12557	91	748

Unsurprisingly the rare events we are attempting to predict (jumps beyond two standard deviations) occur at times when volatility is high. One should be able to produce better predictions than the random solution by limiting positives classifications to times of high volatility. We believe this little solution is doing just that and in doing so highlights the link between news and market volatility. This relationship between news and price volatility is a known one. Vukic [8] shows how the analysis of news split into categories over a year reveals an explicative relationship against the intraday volatility variances of individual components of the French CAC 40. It is probably fair to assume this is a common property of news and markets worldwide.

When we used the *Vix* volatility index as a model input we are unable to replicate the success of the complicated GP solution using news inputs suggesting that the information contribution of news flow data goes beyond that of simple market uncertainty.

6 Conclusion

We have demonstrated that a new and untested data source can give a better prediction of stock price movement than randomness can explain. This is the first published work using this kind of data and we have shown that GP is an appropriate tool to exploit it.

Predictive systems attempt to react to (certain types of) new information faster than the market can. The time it takes for this diffusion of new information is where profit can be make. However an approach that tries to predict movement with fixed times for the look back and look ahead periods is too brittle. As demonstrated, using inputs and outputs that represent larger regions of time, we can give the algorithm a better awareness of the market at each data point and "soften" any inherit assumptions about the news assimilation rate.

From our experiments we believe that this assimilation rate (of news into current price) has a degree of variance to it. Allowing the evolutionary process the freedom to select this rate may be a beneficial avenue of future research. This extra parameter would of course increase the search space exponentially, perhaps the separate input of time data into a Strongly typed GP or GE algorithm could mitigate some of this problem and still achieve the desired goals.

This paper is not concerned with profitability, however it seems feasible that a GP constructed model fed on quantitative news as demonstrated could be coupled with traditional price and volatility inputs to give a market participant a sizable advantage. Further improvements could be made by using advanced trading techniques such as delta hedging and the selective use of the model under conditions when its performance has been shown to exhibit better ratios of true to false positive predictions.

References

1. Kaastra, I., Boyd, M.: Designing a neural network for forecasting financial and economic time series. Neurocomputing 10, 215–236 (1994/1996)
2. Kohzadi, N., Boyd, M.S., Kermanshahi, B., Kaastra, I.: A comparison of artificial neural network and time series models for forecasting commodity prices. Neurocomputing 10, 169–181 (1993/1996)
3. Yao, J., Tan, C.L.: A case study on using neural networks to perform technical forecasting of forex. Neurocomputing 34, 79–98 (1997/2000)
4. Yao, J., Tan, C.L., Poh, H.: Neural Networks for technical analysis: a study on klci. International Journal of Theoretical and Applied Finance 2(2), 221–241(1998/1999)
5. Kaboudan, M.A.: Genetic Programming Prediction of Stock Prices. Computational Economics 16, 207–236 (1999/2000)
6. Becker, Y., Fei, P., Lester, A.M.: Stock Selection - An Innovative application of Genetic Programming Methodology'. In: US Active Equity Research, State Street Global Advisers,
7. Samanta, LeBaron.: Extreme Value Theory and Fat Tails in Equity Markets. In: Computing in Economics and Finance 140, Society for Computational Economics (2005)
8. Vukic, A.: Intraday Public Information The Frence Evidence Thesis (PhD). University of Fribourg (2004)
9. Hodrick and PrescotHodrick, Robert, and E.C. Prescott, Postwar U.S. Business Cycles: An Empirical Investiga-tion, Journal of Money, Credit, and Banking (1997)
10. Lawrenz, C., Westerhoff, D.: Modeling Exchange Rate Behavior with a Genetic Algorithm. Computational Economics 21, 209–229 (2000/2003)

11. Fama, E.: Efficient Capital Markets: A Review of Theory and Empirical Work. Journal of Finance 25, 383–417 (1970)
12. Kelly Jr, J.L.: A New Interpretation of Information Rate, Bell System Technical Journal, vol. 35, pp. 917–926 (1956)
13. Wilmott, P.: Paul Wilmott Introduces Quantitative Finance, 2nd edn. Wiley, John & Sons, Chichester (2006)
14. Tetlock, P.C.: Giving Content to Investor Sentiment: The Role of Media in the Stock Market. Journal of Finance 62, 1139–1168 (2007)

A Genetic Programming Approach to Deriving the Spectral Sensitivity of an Optical System

Marc Ebner

Universität Würzburg, Lehrstuhl für Informatik II
Am Hubland, 97074 Würzburg, Germany
ebner@informatik.uni-wuerzburg.de
http://wwwi2.informatik.uni-wuerzburg.de/staff/ebner/welcome.html

Abstract. In color image processing, several sensors are used which respond to the light in the red, green and blue parts of the spectrum. When working with color images taken by an optical system it is very important to know the sensitivity of the entire optical system. The optical system consists of the sensor, lens and any filters which may be used. The response characteristics of the lens and filters can be measured inside the laboratory. However, for many digital cameras it is not clear how the sensors contained inside the camera respond to light. This information may not be available from the manufacturer of the camera. Even if we knew the response characteristics of the sensor, it may not be clear what algorithms are employed by the manufacturer before the data is finally stored as an image file. We show how genetic programming may be used to obtain the sensor response functions using a single image from a calibration target as input together with the reflectance data of this calibration target.

1 Motivation

The sensor array contained inside a digital camera measures the incident light. For many digital cameras, data about how the sensor responds to light is not publicly available because this data may not be released by the manufacturer. Knowing how the RGB values stored inside the image depend on the irradiance entering the lens of the camera is very important for colorimetry [1,2] and the research area of color constancy [3,4,5,6,7]. We show how genetic programming [8,9,10] can be used to obtain the sensor response functions using an image from a calibration target as input.

A standard sensor consists of a single type of light sensitive sensor and differently colored filters which are placed in front of the sensor to make it respond to light in the red, green and blue parts of the spectrum. These sub-pixel sensors are often arranged in a pattern which is called a Bayer pattern [11]. A full color image is obtained by interpolating the data from adjacent sensors. Other types of sensors where all three components of the incident light are measured at the same position also exist. Imaging chips which measure more than the three components red, green, and blue have also been developed.

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 61–72, 2008.
© Springer-Verlag Berlin Heidelberg 2008

The response of a sensor for a given wavelength is proportional to the irradiance falling onto the sensor times the sensitivity of the sensor for that given wavelength. The energy measured by the sensor is obtained by integrating over all wavelengths. If we know the irradiance falling onto the optical system and also know the energy measured by the sensor, then we can formulate an optimization problem in order to find the sensitivities of the optical system. Data is typically measured at intervals of 10nm. Therefore, we have 32 data points inside the visible range from 390nm to 700nm. Finding the sensitivity of an optical system is basically an optimization problem where the 32 sensitivities at positions $\{390\text{nm}, 400\text{nm}, ..., 700\text{nm}\}$ have to be found.

A number of different problems in image processing have been addressed by the evolutionary computation community. Zhang and Ji [12] as well as Rodehorst and Hellwich [13], have used a genetic algorithm for camera calibration. An evolutionary strategy was used by Cerveri et al. [14] to obtain the internal or external parameters of a camera. Johnson et al. [15] used a genetic algorithm for projector calibration. Carvalho et al. [16] has used a least squares approach to obtain the response function of a sensor. A genetic algorithm was used to maximize the prediction ability of an extended generalized cross-validation measure.

Ebner [17] was the first to apply an evolutionary strategy [18,19] to obtain the sensor response curves of an optical system. Due to the type of problem, constraints have to be enforced in order to solve it. Ebner has shown that best results were obtained by enforcing the constraints directly on the genotype. We will show how genetic programming can be used to find a solution to this type of problem. By properly choosing the set of terminal symbols and the set of elementary functions, constraints are enforced naturally.

This article is structured as follows. First, we describe the model of color image formation. We then explain how finding the response curves of an optical sensor can be defined as an optimization problem. Next, we show how genetic programming can be used to find a solution to this problem. We performed experiments on simulated data where the ground truth is known and also obtained the sensor response curves for two commercially available digital cameras. Conclusions are given at the end of the paper.

2 Theory of Color Image Formation

Suppose that we use our optical system to take an image of a calibration target illuminated by a light source of known spectral power distribution. A calibration target consists of many differently colored patched of known reflectances. The optical system measures the light which is reflected from the calibration target. Let N_p be the number of colored patches on the calibration target. Let $E(p, \lambda)$ be the irradiance which is falling onto patch p at wavelength λ. Some of the irradiance is absorbed by the patch, the remainder is reflected and may enter the lens of the camera. Let $R(p, \lambda)$ be the reflectance of patch p at wavelength λ. We will assume that the optical system is using three sensors which measure the light in the red, green and blue parts of the spectrum. Let $S_i(\lambda)$ be the

sensitivity of the sensor $i \in \{r, g, b\}$. Then the energy $I_i(p)$ measured by sensor i for patch p is modeled as

$$I_i(p) = \int S_i(\lambda) R(p, \lambda) E(p, \lambda) d\lambda. \quad (1)$$

The integration is performed over all wavelengths to which the sensor responds. This model of color image formation is used by many algorithms in colorimetry and color constancy [20,21,22].

We now assume that the calibration target is a Lambertian reflector, i.e. an object which reflects the incident light in all directions. Let the radiance given off by the light source which illuminates the calibration target be $L(\lambda)$. Then the irradiance falling onto the calibration target is simply $E(p, \lambda) = L(\lambda) \cos \alpha$ where α is the angle between the normal vector $\mathbf{n}_S$ describing the orientation of the calibration target and the unit vector $\mathbf{n}_L$ pointing into the direction of the light source from the object patch. Hence, the energy $I_i(p)$ measured by the sensor i for object patch p is given by

$$I_i(p) = G(p) \int S_i(\lambda) R(p, \lambda) L(\lambda) d\lambda \quad (2)$$

where $G(p) = \mathbf{n}_S \mathbf{n}_L^T(p) = \cos \alpha$ is a geometry factor. The geometry factor scales all channels equally.

Digital cameras usually do not save the energy data measured by the sensors. Most produce an output image using the sRGB color space [23]. If the sRGB color space is used, then the measured data is stored in a non-linear way such that the non-linearity of the output device is compensated for. This is called a gamma correction. If we process such images of our calibration target, then this gamma correction needs to be undone such that the processed color data depends linearly on the measured data. Some digital cameras also allow the user to select that the raw measured data be stored in an image file. In this case, the raw data can be processed directly. From now on, we will assume that our optical system produces RGB color triplets c_i as output and that we have $c_i = I_i$.

3 Evolving the Sensitivities of an Optical System

We now show how evolutionary computation can be used to estimate the sensitivities of an optical system. Figure 1 shows the data flow which is used by our system. First an image of a calibration target is taken with the optical system. For our experiments we will be using a standard IT8 calibration target made by Wolf Faust. Such targets are frequently used for calibration of scanners or other optical systems. This calibration target consists of 22×12 colored patches at the top and 24 different gray patches at the bottom. It comes with a complete set of reflectances for each of the patches for wavelengths 390nm to 700nm in steps of 10nm.

Once an image of the calibration target has been taken, the pixel values of each patch are averaged in order to obtain a single color measurement

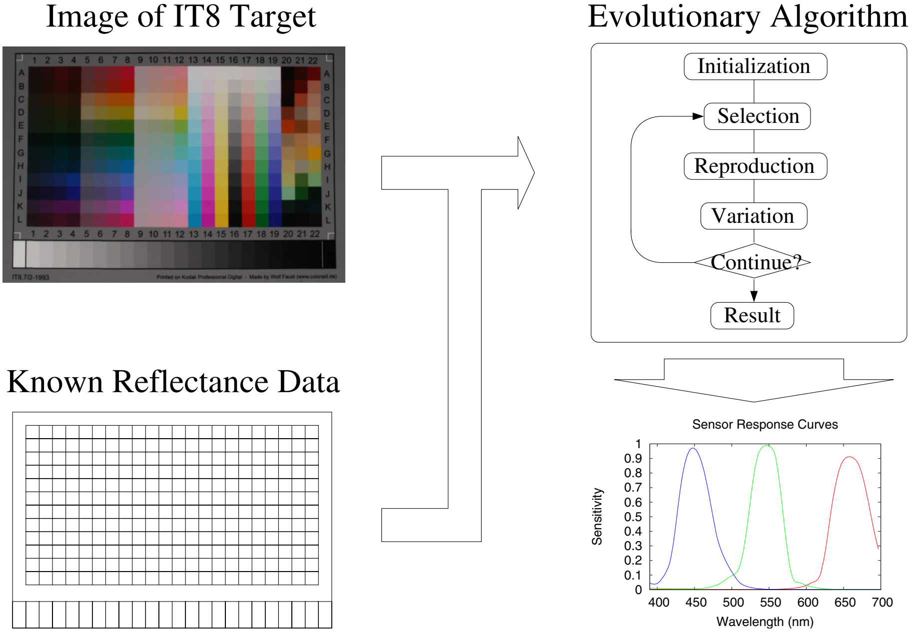


Fig. 1. Data flow of the method to obtain the sensitivities of an optical system. First, the optical system is used to take an image of the calibration target with known reflectances. The reflectance data is used by the evolutionary algorithm to compute the fitness of possible solutions to this problem. After several generations, the optimal sensitivities found by the evolutionary algorithm are output.

$\mathbf{c}(p) = [c_r(p), c_g(p), c_b(p)]$ for each patch p. Pixels close to the border of a patch are not included in the average as they are assumed to be linear combinations of the adjacent colors. Thus, we now have a virtually noise free measurement $\mathbf{c}(p) = \mathbf{I}(p)$ for each patch p. The calibration target comes with known reflectance data $R(p, \lambda)$ for each patch p for each wavelength λ. Before we can solve Equation 2 for $\mathbf{S}(\lambda) = [S_r(\lambda), S_g(\lambda), S_b(\lambda)]$, we also need an estimate of the radiance $L(\lambda)$ which is emitted by the light source. One way to obtain the radiance is to measure it using a spectrometer. Another way is to use a light source which has a known spectral power distribution.

Digital cameras usually perform some kind of white balancing. They correct the image colors for the spectral power distribution of the illuminant. Most consumer cameras either perform automatic white balancing or allow the user to set one of several possible illuminants, such as sun, cloudy sky, neon light, light bulb or flash. Given such a camera, it is best to set the white balance to sun and then take an image of the calibration target on a sunny day. Professional cameras allow the user to choose a particular color temperature. In most cases, it is not known what processing is actually performed inside the camera to obtain the RGB color triplets from the measured data.

Since we have assumed that we took appropriate measures to control the illuminant and that the camera corrects for the type off illuminant used, we now have to solve the following equation to obtain $\mathbf{S}(\lambda)$.

$$\mathbf{c} = G \int \mathbf{S}(\lambda) R(\lambda) d\lambda \tag{3}$$

Note that the geometry factor G scales all color channels equally. It can be removed by computing chromaticities $\hat{\mathbf{c}}$.

$$\hat{\mathbf{c}} = \frac{1}{c_r + c_g + c_b} \mathbf{c} \tag{4}$$

We will be coding the sensor response curves $\mathbf{S}(\lambda)$ as the individuals of our evolutionary algorithm. Given an individual which represents a particular set of sensor response curves, we can then compute how well this set describes the actual set of response curves. In order to determine the fitness of an individual, we compare the measured chromaticities $\hat{\mathbf{c}}_M(p)$ which were obtained from the image of the calibration target with the theoretical chromaticities $\hat{\mathbf{c}}_T(p)$ which are computed using the known reflectances for all patches p.

The known reflectances $R_p(\lambda)$ are used to compute the theoretical chromaticities $\hat{\mathbf{c}}_T(p)$ for patch p. Let $\mathbf{S}(\lambda)$ be the sensor response curve represented by a particular individual. Then the theoretical response is given as

$$\mathbf{c}_T(p) = \sum_{\lambda \in \{390,\ldots,700\}} \mathbf{S}(\lambda) R_p(\lambda). \tag{5}$$

Let $\hat{\mathbf{c}}_T(p)$ be the corresponding chromaticity, i.e.

$$\hat{\mathbf{c}}_T(p) = \frac{1}{\sum_i c_{iT}(p)} \mathbf{c}_T(p). \tag{6}$$

The deviation E_{fit} between the theoretical and the measured response is our error measure

$$E_{\text{fit}} = \sum_p \left(\hat{\mathbf{c}}_T(p) - \hat{\mathbf{c}}_M(p) \right)^2. \tag{7}$$

In other words, we compute the sum of the squared differences between $\hat{\mathbf{c}}_T$ and $\hat{\mathbf{c}}_M$ over all 288 image patches of the calibration target. The error measure E_{fit} describes how well the sensitivities of any given individual match those of the optical system. We want to minimize this error measure. A perfect individual would have $E_{\text{fit}} = 0$.

4 Obtaining the Sensitivities of an Optical System Using Genetic Programming

Ebner [17] has previously used an evolutionary strategy to obtain the sensor response curves $S_i(\lambda)$ which closely match the sensor response curves of an optical

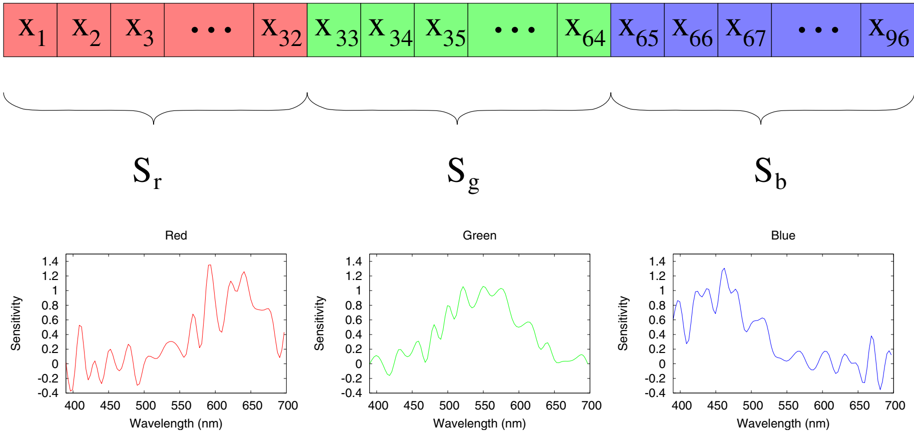


Fig. 2. Evolutionary strategy representation. The sensitivity of the three sub-sensors is stored consecutively inside the genotype. The drawback of this representation is that two adjacent sensitivities are independent from each other leading to a response function which may not necessarily be a smooth response function.

system. An evolutionary strategy is usually used for parameter optimization. For this type of problem, an individual is simply a vector of floating point values which represents the sensitivities of the three sub-sensors at positions $\{390\text{nm}, 400\text{nm}, ..., 700\text{nm}\}$. Such an individual is shown in Figure 2. Due to the type of problem, the search space has to be constrained in order to guide evolution into the correct part of the search space. Here, we have several constraints. The first constraint is that the real sensor response curves are positive for all wavelengths λ, i.e. we have $S_i(\lambda) \geq 0$. Another constraint is that the sensor response curve is smooth without any discontinuities. Due to the computation of chromaticities, we also have the constraint that a uniform scaling of all parameters will not change the result. These constraints can be enforced either through the fitness function or through a repair mechanism on the genotype. Ebner [17] showed that enforcing all the constraints directly on the genotype produced best results.

Instead of encoding an individual as a floating point vector and then enforcing the constraints on the genotype, one may also use a more natural representation for this type of problem. The sensitivity of a sensor is usually Gaussian shaped. One can consider the sensitivity as a combination of Gaussians. This leads us to a genetic programming representation where the terminal symbols are Gaussians which have a particular position and standard deviation inside the visible spectrum and the set of elementary functions simply consists of the addition function. This representation is shown in Figure 3. The nodes are Gaussian functions which depend on the wavelength λ. The internal nodes are used to combine these Gaussian functions.

The set of elementary functions and terminal symbols is shown in Table 1. The terminal symbol $sG(\mu, \sigma)$ computes the following function.

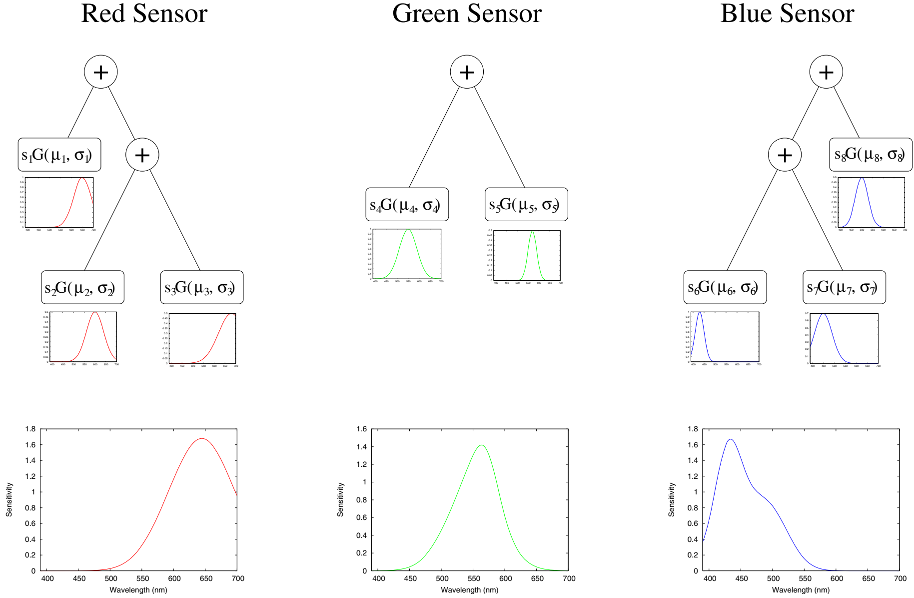


Fig. 3. Genetic programming representation. The response function of a sensor consisting of three sub-sensors responding to light in the red, green, and blue part of the spectrum is represented by three trees.

Table 1. Set of elementary functions and terminal symbols

Name	Symbol	Arity	Internal Variables
Gaussian	G	0	(s, μ, σ)
Addition	$+$	2	none

$$sG(\mu, \sigma) = se^{-\frac{(\lambda-\mu)^2}{2\sigma^2}} \tag{8}$$

The three variables s, μ and σ are stored inside each node. The internal parameter s specifies the strength of the Gaussian, μ specifies its position within the visual spectrum and σ specifies the standard deviation of the Gaussian. Addition is used as the only elementary function. This representation allows us to naturally enforce the constraints. The evolved sensor response curves are simply added Gaussians. Therefore, the evolved sensor response curves are smooth and also fulfill the constraint that the curves are positive for each wavelength λ.

Individuals of the first generation are generated randomly. We then select one of the genetic operators at random. The list of genetic operators are shown in Table 2. Several operators change the structure of the individual, i.e. the trees, while one evolutionary strategy type of mutation operator modifies the internal parameters of all nodes. Offspring are generated until the new population is filled. This process is then iterated for several generations.

Table 2. Genetic programming operators

Name of Operator	Method to generate offspring
Mutation-ES	Evolutionary strategy type of mutation. All nodes of the individual are mutated by adding Gaussian distributed random numbers to the internal parameters. Each internal parameter x has an associated standard deviation δ which is mutated using $\delta := \delta e^{N(0,\tau)}$. The parameter x is then mutated using $x := x + N(0, \delta)$. $N(\mu, \sigma)$ denotes a random number having a normal distribution with mean μ and standard deviation σ.
Mutation-GP	An individual is selected from the parent population. A random node of a random tree of this individual is chosen. Internal nodes are chosen with a probability of 90%. External nodes are chosen with a probability of 10%. A new sub-tree is generated and replaces the chosen node.
Extend-Mutation	An individual is selected from the parent population. A random terminal node of a random tree of this individual is chosen. The chosen terminal node is replaced by the elementary function “Addition”. A new terminal node is generated. The new terminal node and the node that was previously chosen become the child nodes of the newly generated elementary function.
Prune-Mutation	An individual is selected from the parent population. A random terminal node of a random tree of this individual is chosen. The parent node of the chosen terminal node is replaced by the other sub-tree of the parent node. If the tree only consists of a single terminal node then a new terminal node is generated replacing the old one.
Crossover	Two individuals are selected from the parent population. A random sub-tree is selected within the same random tree of both individuals. The two sub-trees are then exchanged between the two individuals. For each crossover, we only generate a single offspring. The second offspring is discarded.
Tree-Crossover	Two individuals are selected from the parent population. We generate one offspring selecting the trees for the offspring from either the first or the second parent.

5 Experiments

A population size of 1000 individuals was used. It was evolved for 1000 generations. Thus, a total of 10^6 fitness evaluations were performed. All individuals from the first generation consisted of three Gaussians (one for each tree) with random positions along the range from $[390, 700]$ and standard deviations from the range $[1, 100]$. An evolutionary strategy type of mutation was used to optimize the strength, the position as well as the standard deviations of all Gaussians of an individual. We are using a standard evolutionary strategy mutation operation with automatic step size adaptation, i.e. each internal parameter has an

associated standard deviation. The mutation step size was initialized to $\sigma = 0.01$ and the variation of the step size was set to 5%, i.e. $\tau = 0.05$. The remaining genetic operators modify the structure of the individual.

The best individual was always reproduced once into the next generation. The remaining individuals of the population were filled using the following percentages: Mutation-ES (90%), Mutation-GP (2%), Extend-Mutation (2%), Prune-Mutation (2%), Crossover (2%), Tree-Crossover (2%). Tournament selection with a tournament size of 5 was used to select individuals. A human would probably approach the problem by first adapting the position and standard deviation of the single Gaussian for each tree and then refining this solution using additional Gaussians as needed. That's why we applied the evolutionary strategy type of mutation much more frequently than the other operators.

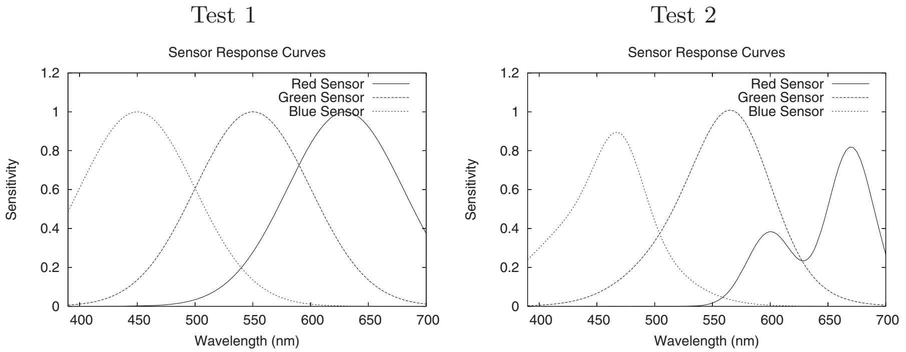


Fig. 4. Three different sensor response functions which are used to evaluate the evolutionary algorithm

We first evaluated the proposed method on two sample problems where the ground truth data is known. We generated synthetic response functions by overlaying Gaussians. These two synthetic response functions are shown in Figure 4. A virtual calibration target with known reflectances was also created. The synthetic response functions were then used to compute the response of the simulated sensor using Equation 5. The evolutionary algorithm evaluates the fitness of an individual using Equation 7. Since we know the actual response function $\mathbf{S}(\lambda)$, we can evaluate how well the evolved response function $\tilde{\mathbf{S}}(\lambda)$ matches this data. For this evaluation, the evolved response function is normalized such that the maximum response is 1.0. The fit to the actual data is then evaluated by computing

$$E_{\text{actual}} = \frac{1}{96} \sum_{\lambda \in \{390,\ldots,700\}} \sum_{i \in \{r,g,b\}} (S_i(\lambda) - \tilde{S}_i(\lambda))^2. \quad (9)$$

The results obtained for both synthetic response functions are shown in Table 3. A total of 10 runs were performed for each sample problem. The table shows the

Table 3. Experimental results obtained during 10 different runs. The standard deviation is shown in round brackets.

Exp	E_{fit}	E_{actual}
Test 1	0.0024(0.0030)	0.0033(0.0092)
Test 2	0.0517(0.0226)	0.0057(0.0024)

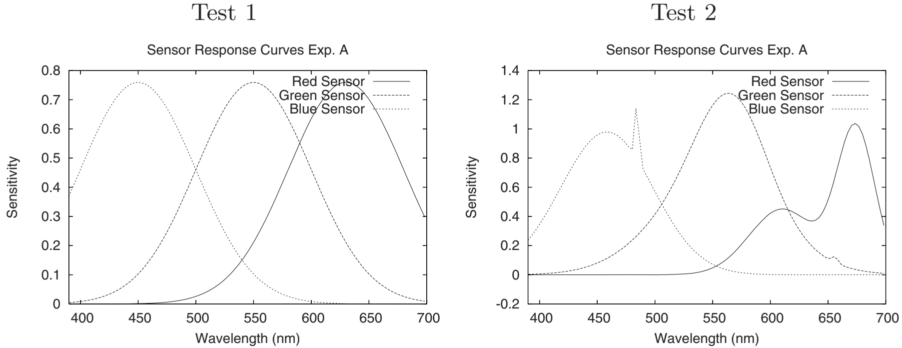


Fig. 5. Best evolved sensor response curves during all 10 runs for the two experiments

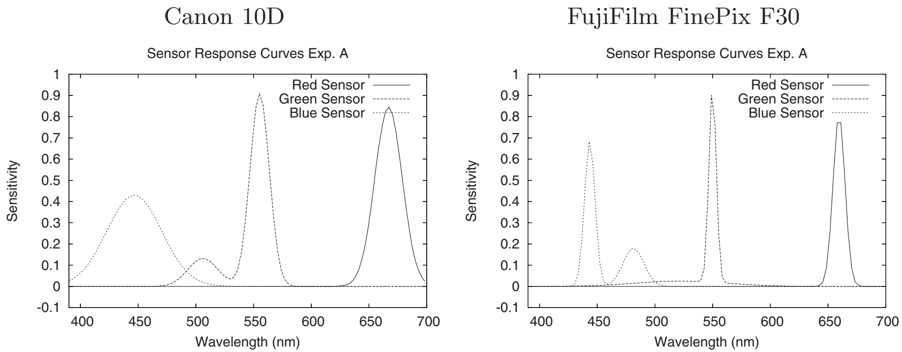


Fig. 6. Best evolved sensor response curves for two commercially available cameras: a Canon 10D with an EF 28-135mm 1:3.5-5.6 IS USM Canon lens and an UV filter and a FujiFilm FinePix F30

average minimum error measure E_{fit} and also the average deviation between the evolved solution and the actual sensor response function E_{actual}. The standard deviations are also shown. The best of the evolved individuals during all 10 runs is shown in Figure 5. The best individuals approximate the actual sensor response curves quite well. However, a problem of this approach is also apparent. Gaussians with a small standard deviation may be introduced which only have a small impact on the fitness of the individual and hence are only eventually removed. At present, it is not clear whether the approach of Ebner [17] or the

approach presented here is better suited to this problem. This will be evaluated in future research.

Apart from testing the proposed method on artificial data, we also used it to obtain the sensitivities of two commercially available digital cameras: a Canon 10D and a FujiFilm FinePix F30. The results obtained are shown in Figure 6.

6 Conclusion

Knowing the spectral sensitivity of an optical system is very important for color vision research. The spectral sensitivities are a result of the type of sensor used and are also influenced by the type of lens and filters which are placed in front of the sensor. We have shown how genetic programming may be applied to this type of problem. The method uses a calibration target with known reflectances. The optical system is used to take an image of the calibration target. Evolution then searches for sensor response curves which reproduce the colors shown in the image of the calibration target. Previously, evolutionary strategies were used to address this problem. Constraints have to be enforced in order to produce a physically plausible sensitivity. This is because the energy measured by a sensor is given by integrating over a range of wavelengths. With our approach the constraints are naturally fulfilled by the type of representation used. We simply represent a sensor response curve as the sum over several Gaussians represented as a tree. The shape of this tree is evolved using genetic programming. Internal parameters which define the position and standard deviations of the Gaussians are evolved using an evolution strategy. We have used two sample problems where the ground truth data is available to evaluate the approach. We then applied this method to obtain the sensor response curves of two commercially available digital cameras.

References

1. Wyszecki, G., Stiles, W.S.: Color Science. Concepts and Methods, Quantitative Data and Formulae, 2nd edn. John Wiley & Sons, Inc, New York (2000)
2. International Commission on Illumination: Colorimetry, 2nd edition, corrected reprint. Technical Report 15.2, International Commission on Illumination (1996)
3. Ebner, M.: Color Constancy. John Wiley & Sons, England (2007)
4. Wandell, B.A.: The synthesis and analysis of color images. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-9(1), 2–13 (1987)
5. Finlayson, G.D., Drew, M.S., Funt, B.V.: Color constancy: generalized diagonal transforms suffice. Journal of the Optical Society of America A 11(11), 3011–3019 (1994)
6. Finlayson, G.D., Hordley, S.D.: Color constancy at a pixel. Journal of the Optical Society of America A 18(2), 253–264 (2001)
7. Finlayson, G.D., Hordley, S.D., Drew, M.S.: Removing shadows from images. In: Heyden, A., Sparr, G., Nielsen, M., Johansen, P. (eds.) ECCV 2002. LNCS, vol. 2353, pp. 823–836. Springer, Heidelberg (2002)

8. Koza, J.R.: Genetic Programming. On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
9. Koza, J.R.: Genetic Programming II. Automatic Discovery of Reusable Programs. MIT Press, Cambridge (1994)
10. 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 Publishers, San Francisco (1998)
11. Bayer, B.E.: Color imaging array. United States Patent No. 3,971,065 (1976)
12. Zhang, Y., Ji, Q.: Camera calibration with gentic algorithms. In: Proceedings of the 2001 IEEE International Conference on Robotics & Automation, Seoul, Korea, May 21-26, IEEE, Los Alamitos (2001)
13. Rodehorst, V., Hellwich, O.: Genetic algorithm sample consensus (gasac) - a parallel strategy for robust parameter estimation. In: International Workshop 25 Years of RANSAC, New York, USA, IEEE, Los Alamitos (2006)
14. Cerveri, P., Pedotti, A., Borghese, N.A.: Combined evolution strategies for dynamic calibration of video-based measurement systems. IEEE Transactions on Evolutionary Computation 5(3), 271–282 (2001)
15. Johnson, C.M., Bhat, A., Thibault, W.C.: An evolutionary approach to camera-based projector calibration. In: Proceedings of the Genetic and Evolutionary Computation Conference 2006, Seattle, Washington, July 8-12, pp. 1871–1872. ACM, New York (2006)
16. Carvalho, P., Santos, A., Dourado, A., Ribeiro, B.: On the estimation of spectral data: a genetic algorithm approach. In: Proceedings of the IEEE International Conference on Image Processing, Thessaloniki, Greece, October 7-10, pp. 866–869. IEEE, Los Alamitos (2001)
17. Ebner, M.: Estimating the spectral sensitivity of a digital sensor using calibration targets. In: Proceedings of the Genetic and Evolutionary Computation Conference, London, England, July 7-11, pp. 642–649. ACM, New York (2007)
18. Rechenberg, I.: Evolutionsstrategie 1994. In: frommann-holzboog, Stuttgart (1994)
19. Schwefel, H.P.: Evolution and Optimum Seeking. John Wiley & Sons, New York (1995)
20. Buchsbaum, G.: A spatial processor model for object colour perception. Journal of the Franklin Institute 310(1), 337–350 (1980)
21. Finlayson, G.D.: Color in perspective. IEEE Transactions on Pattern Analysis and Machine Intelligence 18(10), 1034–1038 (1996)
22. Forsyth, D.A.: A novel approach to colour constancy. In: Second International Conference on Computer Vision (Tampa, FL), December 5-8, pp. 9–18. IEEE Press, Los Alamitos (1988)
23. Stokes, M., Anderson, M., Chandrasekar, S., Motta, R.: A standard default color space for the internet - sRGB. Technical report, Version 1.10 (1996)

A SIMD Interpreter for Genetic Programming on GPU Graphics Cards

W.B. Langdon and Wolfgang Banzhaf

Mathematical and Biological Sciences University of Essex, UK
Computer Science, Memorial University of Newfoundland, Canada

Abstract. Mackey-Glass chaotic time series prediction and nuclear protein classification show the feasibility of evaluating genetic programming populations *directly* on parallel consumer gaming graphics processing units. Using a Linux KDE computer equipped with an nVidia GeForce 8800 GTX graphics processing unit card the C++ SPMD interpretter evolves programs at Giga GP operations per second (895 million GPops). We use the RapidMind general processing on GPU (GPGPU) framework to evaluate an entire population of a quarter of a million individual programs on a non-trivial problem in 4 seconds. An efficient reverse polish notation (RPN) tree based GP is given.

1 Introduction

Whilst modern computer graphics cards deliver extremely high floating point performance for personal computer gaming, the same low cost consumer electronics hardware can be used for desktop (and even laptop) scientific applications [Owens *et al.*, 2007]. However today's GPUs are optimised for a single program multiple data (usually abbreviated Single Instruction Multiple Data SIMD) mode of operation. GPU also place severe limits on data flow. Porting existing applications is non-trivial. Nevertheless [Fok *et al.*, 2007] were able to show speed ups from 0.62 to 5.02 when they ported evolutionary programming to a GPU. They ran EP mutation, selection and fitness calculation on their GPU. Each stage being done by fixed specially hand written GPU programs. [Harding and Banzhaf, 2007] were able to show far higher (peak) speed ups when they ran the fitness evaluation of cartesian genetic programming on a GPU. [Chitty, 2007] used Cg to precompile tree GP programs on the host CPU before transferring them one at a time to a GPU for fitness evaluation. Both groups obtained impressive speed ups by running many test cases in parallel. We demonstrate a SIMD interpreter which runs 204 800 programs simultaneously on the GPU on one or more test cases.

A decade ago [Juille and Pollack, 1996] demonstrated a SIMD GP system for a Maspar MP-2 super computer on a number of problems. The MP-2 was a general purpose supercomputer, costing in the region of $\$10^5$ in the mid 1990s. Its peak theoretical performance came from its many thousands of processing

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 73–85, 2008.
© Springer-Verlag Berlin Heidelberg 2008

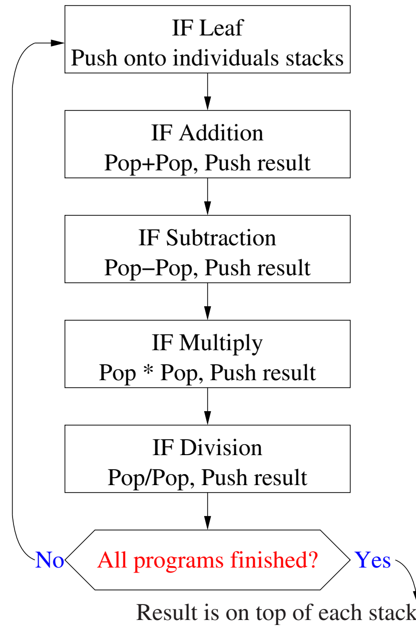


Fig. 1. The SIMD interpreter loops continuously through the whole genetic programming terminal and function sets for everyone in the population. GP individuals select which operations they want as they go past and apply them to their own data and their own stacks.

elements (PE) and the rapid bidirectional 2D data mesh interconnecting them. Jullie's coevolutionary problems were able to exploit the rapid transfer between neighbouring PEs. Less than 200 MP-2 were sold whereas a successful GPU typically has up to 128 independent processors and can be found in literally millions of homes. Even a top of the range GPU can be had for about £350.

In GPUs data describing scenes are imagined to flow into the processors, which transform them and transmit them onto the next processing stage (or to the user's screen). Typically recursion is not used. Part of the GPUs speed comes from specialising this data stream and avoiding the possibility of expensive side-to-side interaction. This restriction enables the GPU to schedule work freely without user intervention between the available processors. Indeed adding more processors can improve performance immediately without redesigning the application. However it makes it difficult to do some operations. The GPU should not be regarded as a "general purpose" computer. Instead it appears to be best to leave some (low overhead) operations to the CPU of the host computer.

Previously the parallelism of GPUs has been exploited by evaluating an individual's fitness by running it simultaneously on multiple training examples. Here we evaluate the entire GP population in parallel. Multiple training examples are not needed. How is this possible on a Single Instruction Multiple Data computer? Essentially the trick is to use one interpreter as the "single instruction"

stream and treat the programs it interprets as "multiple data" items. Figure 1 shows the essential inner loop of the SIMD interpreter. The loop runs on every computing element in the GPU. One complete cycle around the loop is used to evaluate each leaf and function in the GP tree. E.g. five instructions (push $+ - \times$ and $\div$) are needed for each primitive. In the SIMD interpreter, the role of the interpreted data item is to select which of the five is used. (The results of other four are discarded.) Effectively each GP individual acts as a sieve saying which operation it wants performed next. Whilst this introduces a new overhead, use of `cond` instructions to skip the four unwanted instructions and the speed of the GPU makes our approach viable. The SIMD interpreter can support more than four functions, but, in principle, the overhead increases with the size of the function set. While multi-ops, conditionals, loops, jumps, subroutines and recursion are possible, they are not included in these benchmarks.

The next section discusses some other previous parallel GP systems. The section following it discusses possible implementation avenues and why we chose RapidMind. This is followed by descriptions of our two benchmarks (Sections 4 and 5). Whilst Section 6 describes the performance of the interpreter in practise and relates it to other work. This is followed by a discussion, future work (Section 7) and our conclusions (Section 8).

2 Parallel Genetic Programming

While most GP work is conducted on sequential computers, the algorithm typically shares with other evolutionary computation techniques at least three computationally intensive features, which make it well suited to parallel hardware. 1) Individuals are run on multiple independent training examples. 2) The fitness of each individual could be calculated on independent hardware in parallel. 3) Lastly sometimes experimenters wish to assign statistical confidence to the stochastic element of their results. This typically requires multiple independent runs of the GP. The, comparative, ease with which EC can exploit parallel architectures has lead to the expression "embarrassingly parallel".

Early work includes Ian Turton's use of a GP written in Fortran running on a Cray super computer [Turton *et al.*, 1996]. Koza popularised the use of Beowulf workstation clusters where the population is split into separately evolving demes with limited emigration between compute nodes [Andre and Koza, 1996; Bennett III *et al.*, 1999] or workstations [Page *et al.*, 1999]. Indeed as [Chong and Langdon, 1999; Gross *et al.*, 2002] showed by using Java and the Internet, the GP population can be literally spread globally. Alternatively JavaScript can be used to move interactive fitness evaluation to the user's own home but retain elements of a centralised population [Langdon, 2004].

Others have used special purpose hardware. For example, while [Eklund, 2003] used a simulator, he was able to show how a linear machine code GP might be run very quickly on a field programmable gate array using VHDL to model sun spot data. However his FPGA architecture is distant from a GPU.

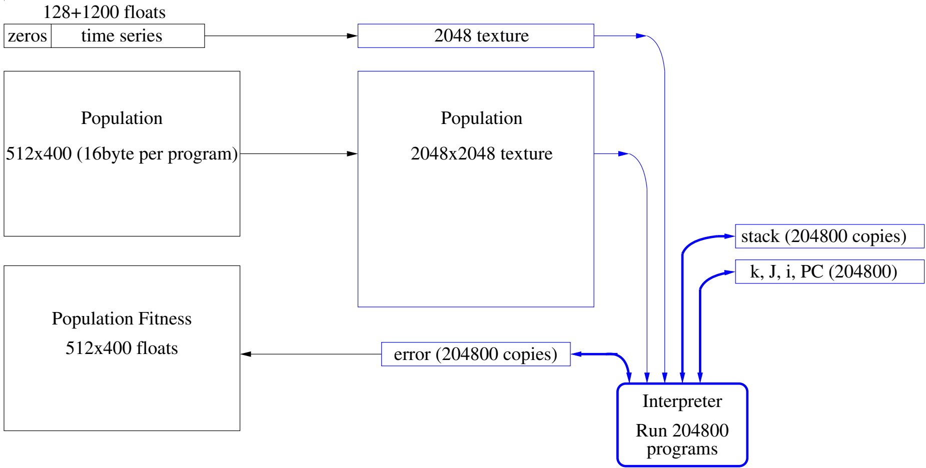


Fig. 2. Major data structures for Mackey-Glass. At the start of the run the interpreter is compiled on the CPU (left hand side). It and the training data are loaded onto the GPU (righthand side). Every generation the whole population is transfered to the GPU. Each individual is interpreted using its own stack and local variables (k, J, i, PC) and its RMS error is calculated. The error is is used as the programs' fitness. All transfers are made automatically by RapidMind.

In summary GP can and has been parallelised in multiple ways to take advantage both of different types of parallel hardware and of different features of particular problem domains. We propose a new way to exploit the inherent parallelism available in modern low cost mass market graphics hardware. Towit a GP SIMD interpreter for GPUs.

3 Programming Graphics Cards

Perhaps unsurprisingly the first uses of graphics processing units (GPUs) with genetic programming were for image generation [Ebner *et al.*, 2005] & its refs.

[Harding and Banzhaf, 2007, Section 3] described the various major high level language tools for programming GPUs (Sh, Brook, PyGPU and microsoft Accelerator). nVidia has two additional tools: CUDA and Cg (C for graphics [Fernando and Kilgard, 2003]). CUDA is specific to nVidia's GPUs. While Sh [McCool and Du Toit, 2004] is still available from SourceForge, its development is effectively frozen at Sh 0.8.0 and McCool recommends using its replacement from RapidMind. Unlike Sh, RapidMind is not free, however www.rapidmind.net issues licences, code, tutorials and documentation to developers. They host a developers' forum and offer prompt and effective support. Like Sh, RapidMind is available for both microsoft directX and unix OpenGL worlds and is not tied to a particular manufacturer's GPU hardware. Indeed recently they started to support parallel programming on the cell processor. However C++ code written for RapidMind's libraries is not portable to other systems. Another nice feature of

RapidMind is that it frees the C++ programmer from the need to learn graphics jargon and conceals many hardware limitations.

4 Mackey-Glass

The Mackey-Glass chaotic time series is described in [Langdon and Banzhaf, 2005b; Langdon and Banzhaf, -]. Briefly the GP is given historical data from a series of 1200 points one time step apart and asked to predict the next value. It is allowed to see data up to 128 time steps in the past. Figure 2 and Table 1 describe our implementation.

Table 1. GPU GP Parameters for Mackey-Glass time series prediction

Function set:	ADD SUB MUL DIV operating on floats
Terminal set:	Registers are initialised with historical values of time series. D128 128 time steps ago, D64 64, D32 32, D16 16, D8 8, D4 4, D2 2 and finally D1 with the previous value. Time points before the start of the series are set to zero (cf. zeros top of Figure 2). Constants 0, 0.01, 0.02, ... 1.27
Fitness:	RMS error
Selection:	fine grained binary tournament demes [Langdon, 1998], non elitist, Population size 512 $\times$ 400
Initial pop:	ramped half-and-half 1:3 (depth 1 to 3. 50% of terminals are constants)
Parameters:	50% subtree crossover. 50% mutation (point 22.5%, constants 22.5%, subtree 5%). Max tree size 15, Max tree depth 4.
Termination:	50 generations

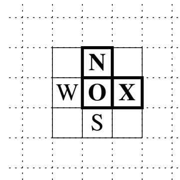


Fig. 3. The GP population is spread one per grid square in two dimensions. If **N**orth is better than **O**rigin, it is copied over it. But if **O**rigin is better, **O** is copied over **N**. (No change if equally fit.) After selection, crossover may occur between **O** and **X**. To promote mixing, 50% of crossovers swap which parent supplies the root node, so a child produced by crossover is equally likely to inherit its root from either parent. Also the neighbourhood pairing rotates 90° every generation. E.g. next generation, crossover will be between **O** and **S**.

4.1 Fine Grained Diffusion Model of Overlapping Demes

While it is not needed for operation with GPU, we used a fine grained diffusion model of overlapping demes [Langdon, 1998], see Figure 3. This allows a low selection pressure and ready visualisation, cf. Figure 4.

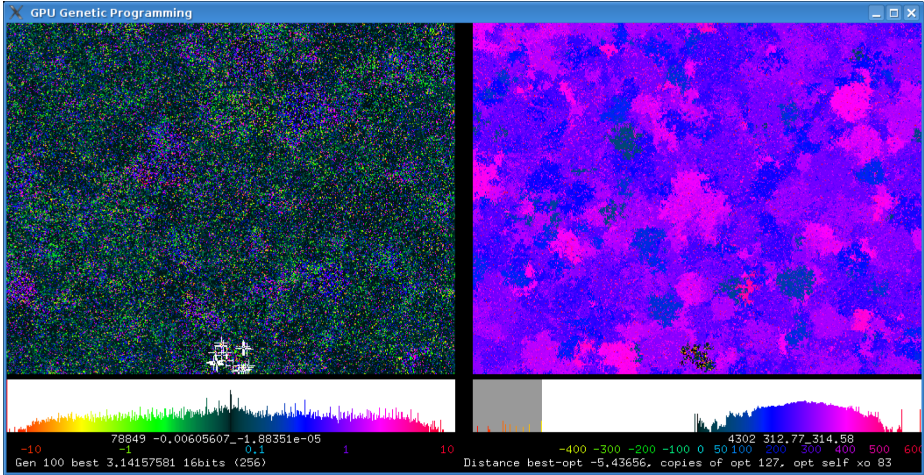


Fig. 4. Screen shot of 512 × 400 GP population after 100 generations. Colour indicates fitness (left) and syntax (right). Below are two histograms (log scale) showing distribution of population by fitness and genotypic distance from the first optimal solution. (Colour scales below each histograms.) Crossover is producing large numbers of unfit leafs (vertical lines at 540 and 600) [Poli *et al.*, 2007]. Local convergence and the production of species is visible (esp. right). See http://www.cs.ucl.ac.uk/staff/W.Langdon/pi2_movie.html and Google videos for animation and more explanation.

4.2 Subtree Crossover and Mutation

In these experiments, the crossover and mutation rates were chosen so that all of the next population are produced either by crossover or mutation (but not both). This ensures almost all children are different from their parents.

Koza's [Koza, 1992] crossover was implemented for linearised reverse polish notation. However there is no bias towards using functions rather than terminals as crossover points. If a pair of crossover points would cause either offspring to be too big or too deep, both are rejected and a new pair chosen again.

One of three types of mutation are used: subtree mutation, point mutation and constant creep mutation. In subtree mutation a subtree is chosen uniformly at random and replaced with a subtree created by the ramped half-and-half (depth 0:1, i.e. leaf or 1 function+2 leafs) algorithm used to create the initial population. If the mutation point is already at the maximum depth, then the subtree is replaced by a randomly chose leaf. If the mutant tree is too big it is rejected and the mutation process restarted with a newly chosen mutation point.

Point mutation does not change the size or shape of the parent tree. A mutation point is uniformly chosen and replaced by a function or leaf with the same arity using the same random selection technique as was used in the initial population. Repeated mutations are applied until, the mutated tree is syntactically different from its parent.

Table 2. Mackey-Glass prediction error after 50 generations in ten runs (multiplied by 128 as was used in [Langdon and Banzhaf, 2005a])

											Mean
RMS error×128	4.69	4.69	4.69	4.69	4.69	4.69	4.69	4.69	4.69	4.69	4.69
Solution size	9	11	9	9	13	9	9	9	9	9	9.6
Run time secs	167.3	168.0	167.5	167.5	167.3	167.4	167.5	167.5	167.5	167.6	167.5

In constant creep mutation, one of the constant leafs in the tree is chosen at random. (If there are no constants, point mutation is used instead.) It is changed by just enough to give the next constant's value. (I.e. by ± 0.01 in the Mackey-Glass experiments).

4.3 Mackey-Glass Model Accuracy

The results of ten independent GP runs on the GPU are summarised in Table 2. The tight limit on tree size (15) and depth (4) lead to similar but smaller solutions than those reported for tree GP [Langdon and Banzhaf, 2005a, , Table 2]. In 4 of 10 cases the results are better than the ten FXO (i.e. the smallest and fastest) subtree runs. The GPU GP runs are faster than all but two CPU runs despite having a population more than 400 times as big and performing full floating point calculations rather than 8 bit integer ones.

Table 3. GPU SIMD GP Parameters for protein localisation

Function set:	ADD SUB MUL DIV operating on floats
Terminal set:	Number (integer) of each of the 20 amino acids in the protein. (Codes B and Z are ambiguous. Counts for code B were split evenly between aspartic acid D and asparagine N. Those for Z, between glutamic acid E and glutamine Q.) 128 unique constants chosen from tangent distribution (50% between -10.0 and 10.0)
Fitness:	$\frac{1}{2}$True Positive rate + $\frac{1}{2}$True Negative rate [Langdon and Barrett, 2004]
Selection:	fine grained binary tournament demes [Langdon, 1998], non elitist, Population size 1024 × 1024
Initial pop:	ramped half-and-half 2:5 (50% of terminals are constants)
Parameters:	50% subtree crossover. 50% mutation (point 22.5%, constants 22.5%, subtree 5%). Max tree size 63, Max tree depth 8.
Termination:	1000 generations

5 Evolving a Million Individuals for 1000 Generations Protein Location Prediction

The system was expanded to cope with: 1) a population of a million programs. 2) bigger trees. 3) deeper trees. 4) Randomised sub-selection of training cases. (See Table 3.) The task chosen was to predict the location of proteins within the cell given only their amino acid composition [Langdon and Banzhaf, -;

Harding and Banzhaf, 2007]. A 1024 by 1024 population of programs of up to 63 tree elements and maximum depth of 8 was run on 200 of 1213 randomly chosen proteins selected for training. Compared to [Langdon and Banzhaf, -, , Table 5], in terms of predictive accuracy on unseen proteins (cf. Figure 5) this run produced better results than one technique (FXO) and the same accuracy but a smaller solution than the other technique (two point crossover, 2XO). However the main point is a graphics card can readily evolve millions of GP programs over thousands of generations.

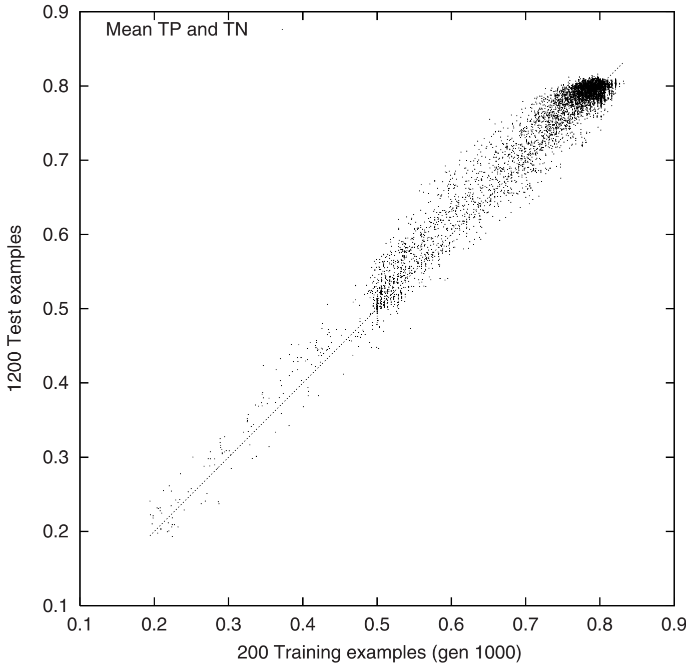


Fig. 5. Fitness on 200 randomly chosen training cases in generation 1000, versus fitness on 1200 unseen proteins. The strong correlations shows GP has learnt for random samples and (better yet) GP models have avoided over fitting.

6 Performance of SIMD Interpreter

6.1 Overhead of Opcode Selection

The interpreter's performance is summarised in Table 4.

We wished to estimate the overhead of the SIMD loop scheduling all of the primitives and then discarding the results of all but the 20% that are needed. To do this we selected a typical evolved Mackey-Glass program and timed how long it took the interpreter to run it. Secondly we hand build an version of the interpreter specific for this program, where every operation is needed and no

Table 4. Speed (millions GP operations/sec) of GPU interpreter on an nVidia GeForce 8800 GTX. Terminal sets $\mathcal{T}$ include inputs and 128 constants

Experiment	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{F}\rvert$	Population	program size	test cases	Speed (M GPops^{-1})
Mackey-Glass	8+128	4	204 800	11.0	1200	895
Mackey-Glass	8+128	4	204 800	13.0	1200	1056
Protein	20+128	4	1 048 576	56.9	200	504
Laser_a	3+128	4	18 225	55.4	151 360	656
Laser_b	9+128	8	5 000	49.6	376 640	190

results are discarded. Rather than the expected five to one ratio, the standard SIMD interpreter is only 2.89 times slower than the specialised one.

A plausible explaination is that: on the GPU floating point operations such as addition and multiplication, which form the GP function set, are extremely fast. It is the GP terminals (which make up 54% of the program) which take longer since they collect the data. The functions only manipulate data already on the stack. This asymmetry in the costs of items in the SIMD dispatch loop means the addition of a few very fast operations has proportionately less impact than was expected.

Potentially this means we could expand the function set to include trigonometry, log, exponentiation, etc. Many of these are directly implemented by the GPU. While increasing the function set would not be free, the additional overhead should be small.

6.2 GPU Speed Up

The Mackey-Glass interpreter was recoded with minimum changes to run on the CPU. A 2211MHz AMD Athlon 64 Processor 3500+ CPU evolved 50 gens of a population of 204 800 trees in 1129.59 seconds. I.e. 7 times longer than the GPU.

7 Discussion

In previous work [Harding and Banzhaf, 2007] used the GPU exclusively for running training cases for cartesian genetic programming and showed impressive speed up in some cases but that improvement was highly variable. Indeed using the GPU was slower than the CPU in a few cases. GP program size and number of training examples per fitness evaluation appear to be critical. We have shown a way of actually executing a traditional tree GP population on the GPU card. It replaces the cost of compiling each member of the population on the CPU by the overhead of running an interpreter on the GPU. Harding's results mostly show that the GPU gives a big performance gain where the compiled GP program is run many times and the programs are large. However if programs are run few times the cost of the compiler and transfer to the GPU may not be repaid. There appears to be a nonlinearity (perhaps in the cost of starting the compiler) so that the relatively small cost of running short programs appears large compared to the cost of compiling them and transferring them to the GPU. With our more

traditional interpreter approach, the population is transfered without compilation overhead to the GPU and the speedup from running in parallel on the GPU appears to be more consistent. We obtain a speed up of more than an order of magnitude for very small programs.

7.1 Implementation Issues

We found that the GPU would give good performance if it was given reasonable chunks of work to do. Say between 1 and 10 seconds. Then the time to transfer the population and the training data into the GPU and fitness vector out is ok.

RapidMind on a Linux platform uses the GNU C++ compiler GCC and GDB debugger. Unfortunately GCC's error reporting can be hard to interpret since RapidMind (like Sh) makes heavy use of templates. RapidMind's cross compiler for the GPU worked seamlessly.

In Sections 4 and 5 the interpreter explicitly loops through all the fitness cases. The GPU can also vectorise computation across cases. We did this recently for a small population and executed 50 000 cases $\times$ pop size in parallel. However, we anticipate it usually remains better to loop through the test cases and so reduce data communication and concentrate computation in fewer threads.

7.2 No Protected Division: Closure

Special cases, like divide by zero, are handled by non data values *nan* and *inf*. The interpreter does not check for divide by zero. Undoubtedly this makes it faster. In effect, the GPU's floating point hardware supplies closure for us.

However we still need to be wary. Potentially large numbers of randomly generated programs, or even offspring of evolved individuals, may have invalid fitness. Filling the population with them may inhibit or even prevent GP successfully evolving.

7.3 Reverse Polish Notation Expression Stack Depth

The GPU does not allow arbitrary *write* access to large arrays. Indeed forcing the data flow out of the GPU to be streamlined is required to enable tasks to be easily shared between the 128 processors and so is partly responsible for the GPUs speed. However it does make it difficult to implement a stack. Therefore it was necessary to simulate a stack using joins. ([Ernst *et al.*, 2004] suggest a somewhat complicated way to implement a GPU stack. It requires at least two passes. [Lefohn *et al.*, 2006] use Cg to efficiently implement a stack. Neither approach is feasible in RapidMind 2.0.1.) Joins work fine for small stacks. Indeed with a stack depth of 4 the interpreter flew at more than a billion GP primitives per second (speed up of more than 12). When the depth was doubled to 8 (for the Mackey-Glass and protein prediction experiments) it imposed about a 30% performance penalty. It appears that a stack limit of 12 or 16 would be feasible. While this may seem restrictive, it is worth remembering that all the original GP experiments [Koza, 1992] were conducted in Lisp with a depth limit of 17.

7.4 Non-tree GP, GP without a Stack

If the compilation overhead is too heavy, our interpreter approach may be attractive. It could be readily applied to cartesian GP [Harding and Banzhaf, 2007] and to linear GP. Typically both approaches use a small number of registers and do not require the use of a stack. Hence a linear genetic program could be interpreted directly on a GPU without incurring the stack overhead or consequent depth limit.

We have deliberately limited ourselves to demonstrating a traditional tree GP actually running on the GPU. We have been prepared to pay the overhead of the instruction loop scheduling one thing at a time. However evolution can often take advantage of muddled situations. We could imagine an evolutionary system in which the program did not wait for exactly the next required instruction to come around. But instead the program could say I will take the result of several instructions, whichever is scheduled first. This might be implemented by the interpreter looking for any bit in a bit mask being set, or an opcode lying in some range, or some other form of fuzzy match between what the program wants and what the interpreter is doing now. Of course the order of the actions of the interpreter might also be evolved. While this form of coevolution is unlikely to yield immediate speed ups on today's problems, it might be a route to meta evolution on more interesting problems in future.

7.5 Possible Extensions

We have shown reliable speed ups can be obtained using a SIMD interpreter to execute a GP population on the GPU. [Fok *et al.*, 2007] have already shown (albeit for EP) that a GPU can implement mutation and selection. Although genetic programming mutation is more complex, we anticipate it too could be implemented on the GPU. Indeed, although [Fok *et al.*, 2007] shied away from crossover, We expect GP crossover could also be performed by the GPU. As GPUs continue to improve, the whole GP may be run by them.

8 Conclusions

By using a postfix (RPN) rather than a prefix (Lisp) representation, we have replaced recursive calls by an explicit stack. Avoiding recursion and using `cond` to select opcodes enabled us to run tree genetic programming with mega populations actually on the GPU. Speed up depends on terminal set, training set size, etc. but parallel operation can yield a speed up of 7–12. Typically a modern GPU interprets hundreds of millions of GP operations per second. Indeed in one case, we exceeded a billion GP ops per second. This is about **0.1 peta GP opcodes per day for $500**.

The SIMD interpreter could be readily adapted to linear GP. Indeed a linear GP system would avoid the overheads associated with simulating a stack. It might be possible to extended it to other types of GP.

C++ code available `ftp://cs.ucl.ac.uk/genetic/gp-code/gpu_gp_1.tar.gz`

Acknowledgements

We would like to thank Simon Harding, Nolan White, Paul Price (MUN) and Joanna Armbruster and Nick Holby. Experiments run at Memorial University.

References

[Andre and Koza, 1996] A parallel implementation of genetic programming that achieves super-linear performance. In: Arabnia, H.R. (ed.) Proc. of Int. Conf. on Parallel and Distributed Processing Techniques and Apps. CSREA, pp. 1163–1174 (1996)

[Bennett III *et al.*, 1999] Building a parallel computer system for $18,000 that performs a half peta-flop per day. In: Banzhaf, et al. (eds.) GECCO 1999, pp. 1484–1490 (1999)

[Chitty, 2007] A data parallel approach to genetic programming using programmable graphics hardware. In: Thierens, D., et al. (eds.) GECCO 2007, pp. 1566–1573. ACM, New York (2007)

[Chong and Langdon, 1999] Java based distributed genetic programming on the internet. In: Banzhaf, et al. (eds.) GECCO 1999, p. 1229 (1999), Full text in CSRP-99-7

[Ebner *et al.*, 2005] Evolution of Vertex and Pixel Shaders. In: Keijzer, M., Tettamanzi, A.G.B., Collet, P., van Hemert, J.I., Tomassini, M. (eds.) EuroGP 2005. LNCS, vol. 3447, pp. 261–270. Springer, Heidelberg (2005)

[Eklund, 2003] Time series forecasting using massively parallel genetic programming. In: Proc. of Parallel and Distributed Processing Int. Symposium, pp. 143–147 (2003)

[Ernst *et al.*, 2004] Stack implementation on programmable graphics hardware. In: Girod, B., et al. (eds.) Proc. Vision, Modeling, and Visualization Conference, pp. 255–262 (2003)

[Fernando and Kilgard, 2003] The Cg Tutorial. Addison-Wesley, Reading (2003)

[Fok *et al.*, 2007] Evolutionary computing on consumer graphics hardware. IEEE Intelligent Systems 22(2), 69–78 (2007)

[Gross *et al.*, 2002] Evolving chess playing programs. In: Langdon, W.B., et al. (eds.) GECCO 2002, pp. 740–747 (2002)

[Harding and Banzhaf, 2007] Fast Genetic Programming on GPUs. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 90–101. Springer, Heidelberg (2007)

[Juille and Pollack, 1996] Massively parallel genetic programming. In: Angeline, P.J., Kinnear Jr., K.E. (eds.) Advances in GP, vol. 2, pp. 339–358. MIT Press, Cambridge (1996)

[Koza, 1992] Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)

[Langdon and Banzhaf, 2005a] Repeated Patterns in Tree Genetic Programming. In: Keijzer, M., Tettamanzi, A.G.B., Collet, P., van Hemert, J.I., Tomassini, M. (eds.) EuroGP 2005. LNCS, vol. 3447, pp. 190–202. Springer, Heidelberg (2005)

[Langdon and Banzhaf, 2005b] Repeated sequences in linear genetic programming genomes. Complex Systems 15(4), 285–306 (2005)

[Langdon and Banzhaf, -] Repeated patterns in genetic programming. In: Natural Computation (2005), doi:10.1007/s11047-007-9038-8

[Langdon and Barrett, 2004] Genetic programming in data mining for drug discovery. In: Ghosh, A., Jain, L.C. (eds.) Evolutionary Computing in Data Mining, pp. 211–235 (2004)

[Langdon, 1998] Genetic Programming and Data Structures. Kluwer, Dordrecht (1998)

[Langdon, 2004] Global Distributed Evolution of L-Systems Fractals. In: Keijzer, M., O'Reilly, U.-M., Lucas, S.M., Costa, E., Soule, T. (eds.) EuroGP 2004. LNCS, vol. 3003, pp. 349–358. Springer, Heidelberg (2004)

[Lefohn *et al.*, 2006] Glift: Generic, efficient, random-access GPU data structures. ACM Transactions on Graphics 25(1), 60–99 (January 2006)

[McCool and Du Toit, 2004] Metaprogramming GPUs with Sh. AK Peters (2004)

[Owens *et al.*, 2007] A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26(1), 80–113 (2007)

[Page *et al.*, 1999] Smooth Uniform Crossover with Smooth Point Mutation in Genetic Programming: A Preliminary Study. In: Langdon, W.B., Fogarty, T.C., Nordin, P., Poli, R. (eds.) EuroGP 1999. LNCS, vol. 1598, Springer, Heidelberg (1999)

[Poli *et al.*, 2007] On the Limiting Distribution of Program Sizes in Tree-Based Genetic Programming. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 193–204. Springer, Heidelberg (2007)

[Turton *et al.*, 1996] Some geographic applications of genetic programming on the Cray T3D supercomputer. In: Jesshope, C., Shafarenko, A. (eds.) UK Parallel 1996, Springer, Heidelberg (1996)

Partitioned Incremental Evolution of Hardware Using Genetic Programming

David Jackson

Dept. of Computer Science, University of Liverpool
Liverpool L69 3BX, United Kingdom
djackson@liverpool.ac.uk

Abstract. In an effort to enable evolutionary computation techniques to discover solutions for large and complex hardware systems, techniques have been devised to break the initial problem down into smaller sub-tasks. In particular, a decomposition approach has been described that is based on partitioning of the circuit test vectors, but it has its limitations. In an effort to address this, we have combined the partitioning method with an incrementally evolving genetic programming approach. The result, referred to as Partitioned Incremental Evolution of HARDware (PIE-HARD), exhibits solution-finding performance that is significantly better than that of other approaches.

1 Introduction

Recent years have seen a growth in interest in the application of evolutionary computation techniques to the design of hardware systems, both analogue [1-3] and digital [4-8]. Such evolvable hardware approaches may be divided into two camps: intrinsic, in which evolved circuit designs are implemented and evaluated on hardware itself; and extrinsic, where assessment is carried out via software emulation. Whichever route is taken, a key problem is that of scaling up the techniques to deal with large, complex circuits.

One approach is to take an algorithmic view of hardware, which in turn opens up the possibility of using genetic programming (GP) as a means for evolving circuits. One of the advantages of using GP in this way is that it is based on arbitrary length chromosomes, so that the number of gates used to implement a circuit is not confined to a pre-determined size. Another advantage is that GP has been extensively used as a vehicle for investigating decomposition and module reuse when solving complex problems, giving rise to a number of techniques that may be of value in the hardware arena.

Within the GP field, the most well-known approach to hierarchical evolution is Koza's automatically defined functions (ADFs) [9-11]. 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 [12]) have provided extensive evidence that problems are often solved more readily by using ADFs than they are without them.

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 86–97, 2008.
© Springer-Verlag Berlin Heidelberg 2008

Alternatives to the use of ADFs include the technique of Module Acquisition introduced by Angeline and Pollack [13, 14]; the Adaptive Representation through Learning (ARL) algorithm of Rosca and Ballard [15]; the module selection scheme of Roberts et al [16]; and the encapsulation technique employed by Walker and Miller in their work on Cartesian Genetic Programming [17]. In a more hardware-oriented context, Lopez et al describe the reuse of GP code segments in evolving adders and multipliers [18].

All this is not to say that hardware evolution techniques outside the realm of GP have ignored the value of decomposition and reuse in circuit design. In particular, Torresen [5, 6] has proposed a promising decomposition approach based on partitioning of a circuit's test vectors. We shall return to this in Section 2, where as well as describing the approach in more detail we will also point out some of its limitations.

In Section 3 we describe one of our own contributions to the introduction of hierarchy within GP when solving complex problems. Rather than taking a structural approach as most existing systems have done, we instead use a system based on learning, in which the genetic material used to solve a comparatively simple problem is used as the basis for solving higher-level, more complex problems. We will show how this technique applies particularly well to simple single-output circuits, but that it also has drawbacks for multi-output logic.

In Section 4 we explain how each of the approaches discussed in Sections 2 and 3 addresses the weaknesses of the other. As a consequence, we propose a system based on a combination of the two, which we refer to as Partitioned Incremental Evolution of HARDware (PIE-HARD). The effectiveness of the approach is also evaluated experimentally in that Section. Finally, Section 5 presents some conclusions and pointers to further work.

2 Vector Partitioning

The divide-and-conquer approach to hardware evolution as described by Torresen [5, 6] involves dividing the input test vectors into partitions, and then evolving solutions for each of these partitions independently. In this way, the original system is divided into a number of simpler hardware subsystems which ought to be easier to evolve. Fig. 1 shows how this might be done to evolve a logic circuit for a pattern recognizer. In this example, a pattern represented as an 8-bit binary value is to be classified into one of 4 possible categories A, B, C or D. Hence, the circuit takes 8 inputs and produces 4 outputs. For any test pattern, one of the 4 outputs should be set at logic one to indicate the category, and all other outputs should be logic zero.

The partitioning method considers all of the inputs but only a subset of the available outputs. As shown in Fig. 1, one possibility is to use just two partitions: the first involving only A and B patterns, the second involving only C and D patterns. Thus we now have two subsystems, each with 8 inputs but only 2 outputs. Because each subsystem is less complex than the original circuit, it should be easier to evolve, and each subsystem can be evolved independently of the others. Creating a circuit to solve the original problem involves putting the subsystems together in such a way that all inputs are presented to all subsystems in parallel, although in principle it would be possible to use the subsystems as building blocks in a subsequent evolutionary process to integrate them more tightly.

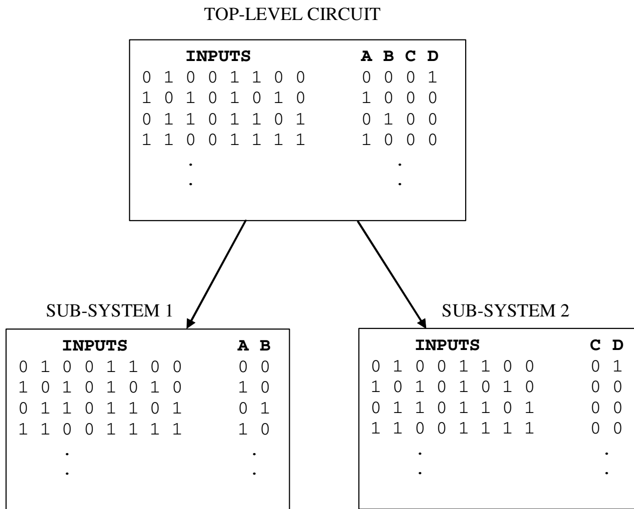


Fig. 1. Example of the vector partitioning method applied to the evolution of a simple classifier circuit. Each simplified unit handles only a subset of the available outputs.

The number of partitions used is flexible. In our example classifier, it would also be possible to evolve four subsystems, each producing just one of the four outputs. In Torresen's experiments, which included a pattern classifier similar to the one we have described, it was found that the partitioning approach offered significant performance improvements over the standard approach of attempting to evolve the high-level circuit directly.

There are however, a number of disadvantages and limitations of the partitioning approach. Firstly, its scope is limited by the number of outputs present in the top-level circuit. In the extreme case, if this circuit has only one output, then no partitioning in the manner described can be done. Secondly, the technique does nothing to reduce the number of inputs entering each subsystem. Each subsystem has exactly the same number of inputs as in the original, top level circuit. If there are n binary inputs, then a subsystem has to pass 2^n test cases in order to be considered a solution, and each additional input doubles this number. Although it is hoped that partitioning will reduce the complexity of each subsystem, the number of test cases that it must satisfy may make it extremely difficult to evolve it even if partitioning is so fine-grained that it is responsible for only one output. Moreover, any individual that is evaluated via consideration of these test cases is only one small part of the overall circuit, and the individuals involved in the evolution of the other subsystems must be similarly assessed. Again, partitioning should lead to reduced complexity in the subsystems, but since there is no shared logic the total computation effort may still be considerable.

3 Layered Learning

In Layered Learning Genetic Programming (LLGP) [19-21], evolution begins in an initial layer that proceeds towards the solution of a sub-task of the overall problem.

The genetic material produced at the end of this layer then forms the basis for the initial population of the next layer, which uses what has been 'learnt' in layer 1 to help it evolve towards a solution of the original problem.

There are various ways of specifying the initial decomposition. In the one we use here, the lower layer is used to evolve a simpler, lower-order version of the original problem. The issue being addressed is thus one of scalability, the original problem being scaled down in the hope of producing genetic material that will be of use in solving the more difficult version. In hardware design, complex units are often formed by connecting simpler but functionally similar sub-units, and adopting a similar approach in evolutionary systems seems intuitively promising.

In our approach [21], genetic material is passed between layers in the form of parameterised modules. 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. 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 branches of the programs, which are free to make use of the function as evolution proceeds towards generating a solution to the higher-order problem.

Table 1. GP parameters for the even-4 parity problem

Objective	To evolve a program capable of determining if the number of logic 1s on the 4 inputs is even
Terminal set	D0, D1, D2, D3
Function set	AND, OR, NAND, NOR
Initial population	Ramped half-and-half
Evolutionary process	Steady-state; 5-candidate tournament selection
Fitness cases	16, representing all combinations of inputs
Fitness	Number of mismatches with expected outputs (0-16)
Success predicate	Zero fitness (solution found)
Other parameters	Pop size=500; Gens=51; prob. crossover=0.9; no mutation; prob. internal node used as crossover point=0.9

As an example, consider the problem of evolving a digital logic circuit which returns a TRUE output if the number of logic one values on its 4 inputs is even, FALSE otherwise. This is the well-known even-4 parity problem. In solving it with standard genetic programming, the parameters we use are presented in Table 1. As an additional bench- mark against which to test the effectiveness of our layered learning approach, we have also used GP systems which allow automatically defined functions (ADFs) to be evolved. In implementing this, we have followed Koza's precept [9] 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).

Table 2. 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

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 [9], defined as the minimum number of individuals that must be processed to achieve a 0.99 probability that a solution will be found. Table 2 compares the performance figures for each of the systems we have described. In the case of the layered learning system, two versions were produced: 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 3. 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

It is clear that the layered learning approach dramatically out-performs conventional and ADF-based GP. Similar improvements are found in higher-order versions of the problem. Table 3, for example, shows the results obtained for the even-6 parity problem, with the population size increased from 500 to 2000. As before, both versions of the layered learning approach perform much better.

Despite these encouraging results, there are limitations to the layered approach as we have outlined it. A particular source of difficulty is the fact that many hardware systems produce multiple outputs. LLGP works by encapsulating a solution to a lower-order version of the problem as a single parameterised module which can be invoked by a separately evolved main branch. However, GP functions normally return only one result. Requiring a module to return multiple results introduces complications. It would be possible, for example, to have the module combine the separate outputs together into a single multi-bit result, but then additional functions are required both to join signals together and then to split them again in the main branch. Other possibilities are to make use of globally accessible variables, or to use pass-by-reference when invoking the module. However, these mechanisms also add complexity to the GP system. Moreover, none of the suggested approaches removes the burden from the lower layer module of having to evolve logic for multiple outputs simultaneously.

4 Partitioned Incremental Evolution

The previous two sections of the paper have described: (i) a partitioning approach which provides benefits by decomposing a circuit based on subsets of the available outputs, but which cannot offer gains for single-output logic; and (ii) an incremental evolution technique which can boost evolution performance for single-output hardware designs, but which does not extend well to multiple outputs. It therefore seems appropriate to attempt to bring together the complementary strengths of the two approaches in a single hardware evolution system. Henceforth, we shall refer to this new system as PIE-HARD (Partitioned Incremental Evolution of HARDware).

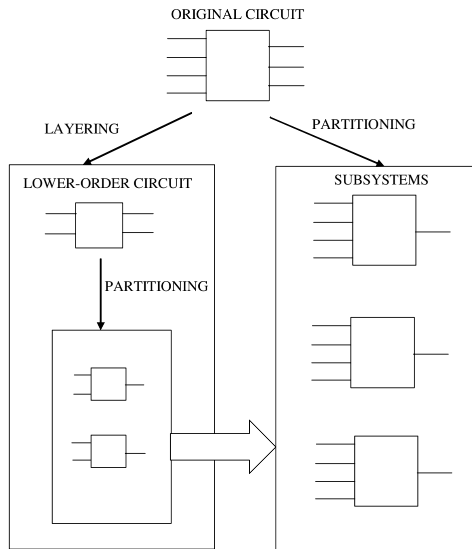


Fig. 2. Overview of the PIE-HARD approach, showing how partitioning is used in both layers, the components of the lower layer feeding into the evolution of the upper

An overview of the PIE-HARD approach is given in Fig. 2. To evolve hardware to realise the original, top-level circuit, we use vector partitioning to divide it into a number of subsystems, as shown on the right hand side of the diagram. However, unlike the approach described in Section 2, we do not try to evolve those subsystems directly. Instead, we firstly solve a lower order version of the original problem. To do so, we also divide the lower-level circuit via vector partitioning into a number of single-output parameterised modules. Genetic programming is then used to evolve solutions for each of these modules in turn. Once all of the modules have been created, they are made available to be called as functions during the next stage of evolution, which aims to solve each of the higher-order subsystems.

In the following experiments, we will show exactly how PIE-HARD operates for realistic circuits.

4.1 Half Adder

In a half-adder circuit, two numerical values are added to produce a result. In its simplest form, each of the inputs is a single binary digit, and there are two output signals which together encode the possible summation results 0-2. The most significant of these two output bits is often referred to as the carry-out.

In our first set of experiments we will attempt to evolve a 2-bit half adder. That is, each of the two addends comprises 2 bits, representing the values 0-3, and the output consists of three bits, capable of holding the results 0-6. In evolving this circuit via standard GP, most of the parameters remain as given earlier in Table 1. The terminal set is now {A0, A1, B0, B1} for the four input lines, and the function set is {AND, OR, NOT, XOR}. To enable conventional GP to deal with the multiple outputs required, we also define a pseudo-function called JOIN. This is present only at the root node of each individual and acts simply as a connector for the multiple branches corresponding to the circuit's outputs. As in the even-4 parity example, we will also use an ADF-based GP system for comparison; for a 2-bit adder with 4 inputs, each individual will have 2 ADFs – one with arity 2 and the other with arity 3.

For further comparison, we will also examine the effectiveness of using vector partitioning in isolation (i.e. without the incremental evolution offered by a layered learning approach). For a 2-bit half adder with 4 inputs and 3 outputs this gives us three subsystems, each with 4 inputs and one output. This is precisely the situation depicted in Fig. 2.

In using PIE-HARD, we need to introduce a lower-order version of the problem in the bottom layer. Since we are solving a 2-bit adder, the lower-order problem needs to be a 1-bit adder. Vector partitioning of this sub-problem leads to 2 modules, each with 2 inputs and one output. Again, this is exactly as shown in Fig. 2. PIE-HARD's first task, then, is to evolve code for each of the modules forming the sub-divided logic of the 1-bit adder. Once these have emerged, they are made available as callable functions in the code being evolved to solve the subsystems of the 2-bit adder.

Table 4 presents the performance figures for each of the systems we have described. Again, where evolution consists of multiple stages (as in vector partitioning and PIE-HARD) the computational effort figure is over all stages.

As can be seen, the introduction of ADFs does nothing to aid the performance of a standard GP system. When vector partitioning is used there is an enormous step up in performance, but PIE-HARD improves on this even further, cutting the computational effort by more than half.

Table 4. Performance figures for evolution of a 2-bit half-adder

Approach	Success rate (%)	Comp. Effort
Monolithic GP	13	490,000
Monolithic GP with ADFs	4	2,881,500
Vector partitioning	68	44,000
PIE-HARD	85	20,000

For a 3-bit adder, the task becomes more difficult. The expansion of the terminal set to {A0, A1, A2, B0, B1, B2} effectively quadruples the number of test cases to be satisfied. However, it also offers us a choice in the case of PIE-HARD, in that either a 1-bit adder or a 2-bit adder can form the basis for the lower layer. Table 5 gives the comparative figures for both of these options. In all the systems in this experiment, the population size is set at 2000.

Table 5. Performance figures for evolution of a 3-bit half-adder

Approach	Success rate (%)	Comp. Effort
Monolithic GP	0	-
Monolithic GP with ADFs	0	-
Vector partitioning	6	4,800,000
PIE-HARD using 1-bit	16	1,674,000
PIE-HARD using 2-bit	50	432,000

The conventional GP approach, either with or without ADFs, is unable to evolve a 3-bit adder circuit. Vector partitioning does at least find some solutions, but is clearly struggling. Although it often manages to evolve some of the subsystems, it rarely manages to discover them all within 50 generations. PIE-HARD again comes out on top, particularly so when it solves a 2-bit adder in its lower layer, leading to half of all runs finding solutions and a computational effort that is less than a tenth of that needed in the partitioning system.

4.2 Full Adder

A full-adder circuit requires an extra input referred to as the carry-in. The idea is that the carry-out signal from one adder unit becomes the carry-in of the next unit, allowing adders of larger bit-widths to be constructed. In one sense this makes evolution more difficult, since the additional input doubles the number of test cases. On the other hand, it offers evolutionary systems, especially hierarchical variants, the opportunity to discover ways of combining sub-units via the connection of carry signals in the manner just described.

Beginning again with a 2-bit adder, we achieve the results shown in Table 6. The terminal set is {A0, A1, B0, B1, CIN}, and the population size has been set at 1000.

Table 6. Performance figures for evolution of a 2-bit full-adder

Approach	Success rate (%)	Comp. Effort
Monolithic GP	5	3,240,000
Monolithic GP with ADFs	1	18,819,000
Vector partitioning	31	572,000
PIE-HARD	97	20,000

The extra input and corresponding increase in test cases has made it difficult for standard and ADF-based GP to cope. Partitioning into three subsystems improves matters, but each subsystem still has to solve for 32 test cases; hence, this approach does not do nearly so well as it did for a 2-bit half-adder. By contrast, the possibility of connecting simpler sub-units via their carry signals appears to have been successfully discovered and exploited by PIE-HARD, giving it a success rate that is over three times greater than that of the partitioning approach, and a hugely reduced computational effort.

Progressing to a 3-bit version, we have again tried PIE-HARD with both 1-bit and 2-bit adders being evolved in the lower layer. The results for a population size of 2000 are presented in Table 7. This time the vector partitioning approach joins the conventional GP systems in being unable to evolve a 3-bit full-adder. PIE-HARD is much more successful, although this time it is the version which evolves a 1-bit adder in the lower layer that exhibits the best performance.

Table 7. Performance figures for evolution of a 3-bit full-adder

Approach	Success rate (%)	Comp. Effort
Monolithic GP	0	-
Monolithic GP with ADFs	0	-
Vector partitioning	0	-
PIE-HARD using 1-bit	65	308,000
PIE-HARD using 2-bit	46	720,000

4.3 Counter

Moving away from addition circuits, our next example is that of a counter circuit. In this, a count of the logic one values present on the data lines is output as a multi-bit integer. For example, if there are ten data inputs, then four output lines will be needed to represent each of the possible counts from 0 to 10 as a binary number. The function set for the problem is {AND, OR, NOT}. Other evolutionary parameters are as those used for the even-parity problem in Table 1. Because of the poor showing of the ADF approach in previous experiments, that technique has been abandoned here.

Beginning with a 3-input counter and a population size of 500, the comparative results are given in Table 8. The PIE-HARD system evolves a 2-bit counter in its lower layer.

Table 8. Performance figures for evolution of a 3-input counter

Approach	Success rate (%)	Comp. Effort
Monolithic GP	22	300,000
Vector partitioning	72	44,000
PIE-HARD	100	2,500

Although it is a reasonably difficult problem for standard GP to solve, the division of the 2-output circuit into two halves makes it much easier. However, once again it is the PIE-HARD approach which has the greatest success: every run generates a solution and within only a few generations, resulting in a comparatively tiny computational effort.

Moving to a 4-bit counter requires that the circuit has three outputs. With the population doubled to 1000 individuals, the results for the various approaches are shown in Table 9. The PIE-HARD system is tried with a 2-bit counter and a 3-bit counter being evolved in the initial layer.

Table 9. Performance figures for evolution of a 4-input counter

Approach	Success rate (%)	Comp. Effort
Monolithic GP	0	-
Vector partitioning	0	-
PIE-HARD using 2-bit	76	60,000
PIE-HARD using 3-bit	23	406,000

The additional input and output make all the difference as far as the vector partitioning system goes, it being now unable to find any solutions. For PIE-HARD, the evolution of a 2-bit counter in the lower layer leads to overwhelmingly better performance than using a 3-bit counter.

These results suggested that it might be possible for the PIE-HARD approach to evolve a 5-bit counter whilst maintaining the population size at 1000. Table 10 bears this out. Note that this time it is the evolution of a 3-bit counter in the lower layer that leads to the better performance.

Table 10. Performance of PIE-HARD in evolving a 5-input counter

Approach	Success rate (%)	Comp. Effort
PIE-HARD using 2-bit	8	2,550,000
PIE-HARD using 3-bit	33	442,000

5 Conclusions

In this paper we have firstly examined two ways of simplifying the task of hardware evolution by introducing a hierarchical approach. The first – partitioning of the circuit into subsystems responsible for specific subsets of the circuit outputs – has proved to be effective for a number of problems. It is, however, limited by the number of outputs available, and it does not reduce the number of test cases that must be considered during fitness evaluation; overall, in fact, the number of subsystems acts as a multiplier of the test cases to be applied to solve the original problem.

The second approach uses learning rather than structure as the basis for decomposition. Systems based on this approach firstly solve a simpler version of the problem, and then use the genetic material evolved as the foundation for evolving a solution to the original problem. This incremental approach has proved to be highly effective in evolving single-output digital logic designs. Its use of parameterised modules does mean, however, that it does not extend well to multi-output logic.

By bringing the two approaches together in PIE-HARD, we produce a system in which each of the two approaches just outlined addresses the weaknesses of the other. As we have shown, this hybrid system exhibits a solution-finding performance that is significantly better than that of other approaches.

There are many opportunities for further work arising from the research described here. For instance, it is not yet clear how wide-ranging the approach is: although it is easy to see that a lower-order version of a 3-bit adder is a 2-bit adder, it is not immediately obvious what is meant by a lower-order version of, say, a prosthetic hand controller. Another unanswered question regarding the lower order problem is the level of simplicity that should be addressed. In solving a 4-bit counter circuit, the use of a 2-bit counter in the lower layer proved more advantageous than using a 3-bit counter, but the reverse was true in our experiments on a 5-bit counter. The reason for this may be something as straightforward as the number of bits being odd or even, but further analysis is necessary to confirm or deny that and to give clues as to how layering should be implemented in the more general case. With regard to this, another avenue of research would be to attempt to extend our dual layer approach to multiple layers; whilst introducing more steps into the problem-solving process, it should also have the effect of smoothing the evolutionary gradient, and it would be interesting to see the results of the trade-offs involved.

References

1. Koza, J.R., Bennett, I.F.H., Andre, D., Keane, M.A., Dunlap, F.: Automated Synthesis of Analog Electrical Circuits by Means of Genetic Programming. IEEE Trans. Evol. Comput. 1(2), 109–128 (1997)
2. Thompson, A., Layzell, P., Zebulum, R.S.: Explorations in Design Space: Unconventional Electronics Design through Artificial Evolution. IEEE Trans. Evol. Comput. 3(3), 167–196 (1999)
3. Alpaydin, G., Balkir, S., Dundar, G.: An Evolutionary Approach to Automatic Synthesis of High-Performance Analog Integrated Circuits. IEEE Trans. Evol. Comput. 7(3), 240–252 (2003)
4. Miller, J.F., Job, D., Vassilev, V.K.: Principles in The Evolutionary Design of Digital Circuits – Part I. Genetic Programming and Evolvable Machines 1, 7–35 (2000)
5. Torresen, J.: A Scalable Approach to Evolvable Hardware. Genetic Programming and Evolvable Machines 3, 259–282 (2002)
6. Torresen, J.: A Divide-and-Conquer Approach to Evolvable Hardware. In: Sipper, M., Mange, D., Pérez-Uribe, A. (eds.) ICES 1998. LNCS, vol. 1478, Springer, Heidelberg (1998)
7. Coello, C.A.C., Christiansen, A.D., Aguirre, A.H.: Use of Evolutionary Techniques to Automate the Design of Combinational Circuits. International Journal of Smart Engineering System Design 2(4), 229–314 (2000)
8. Coello, C.A.C., Luna, E.H., Aguirre, A.H.: Use of Particle Swarm Optimization to Design Combinational Logic Circuits. In: Tyrrell, A.M., Haddow, P.C., Torresen, J. (eds.) ICES 2003. LNCS, vol. 2606, pp. 398–409. Springer, Heidelberg (2003)
9. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
10. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge (1994)

11. Koza, J.R.: Simultaneous Discovery of Reusable Detectors and Subroutines Using Genetic Programming. In: Proc. 5th Int. Conf. Genetic Algorithms (ICGA-1993), pp. 295–302 (1993)
12. Rosca, J.P., Ballard, D.H.: Hierarchical Self-Organization in Genetic Programming. In: Proc 11th International Conf. on Machine Learning, pp. 251–258. Morgan Kaufmann, San Francisco (1994)
13. Angeline, P.J., Pollack, J.: Evolutionary Module Acquisition. In: Proc. 2nd Annual Conf. on Evolutionary Programming, La Jolla, CA, pp. 154–163 (1993)
14. Angeline, P.J., Pollack, J.: Coevolving High-Level Representations. In: Langton, C.G. (ed.) Artificial Life III, pp. 55–71. Addison-Wesley, Reading (1994)
15. Rosca, J.P., Ballard, D.H.: Discovery of Subroutines in Genetic Programming. In: Angeline, P., Kinnear Jr., K.E. (eds.) Advances in Genetic Programming 2, ch. 9, pp. 177–202. MIT Press, Cambridge (1996)
16. Roberts, S.C., Howard, D., Koza, J.R.: Evolving Modules in Genetic Programming by Subtree Encapsulation. In: Miller, J., Tomassini, M., Lanzi, P.L., Ryan, C., Tetamanzi, A.G.B., Langdon, W.B. (eds.) EuroGP 2001. LNCS, vol. 2038, pp. 160–175. Springer, Heidelberg (2001)
17. Walker, J.A., Miller, J.F.: Evolution and Acquisition of Modules in Cartesian Genetic Programming. In: Keijzer, M., O'Reilly, U.-M., Lucas, S.M., Costa, E., Soule, T. (eds.) EuroGP 2004. LNCS, vol. 3003, pp. 187–197. Springer, Heidelberg (2004)
18. Lopez, E.G., Poli, R., Coello, C.A.C.: Reusing Code in Genetic Programming. In: Keijzer, M., O'Reilly, U.-M., Lucas, S.M., Costa, E., Soule, T. (eds.) EuroGP 2004. LNCS, vol. 3003, pp. 359–368. Springer, Heidelberg (2004)
19. 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)
20. Hsu, W.H., Harmon, S.J., Rodriguez, E., Zhong, C.: Empirical Comparison of Incremental Reuse Strategies in Genetic Programming for Keep-Away Soccer. In: Deb, K., et al. (eds.) GECCO 2004. LNCS, vol. 3102, Springer, Heidelberg (2004)
21. Jackson, D., Gibbons, A.P.: Layered Learning in Boolean GP Problems. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 148–159. Springer, Heidelberg (2007)

Population Parallel GP on the G80 GPU

Denis Robilliard, Virginie Marion-Poty, 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,fonlupt}@lil.univ-littoral.fr

Abstract. The availability of low cost powerful parallel graphics cards has stimulated a trend to port GP on Graphics Processing Units (GPUs). Previous works on GPUs have shown evaluation phase speedups for large training cases sets. Using the CUDA language on the G80 GPU, we show it is possible to efficiently interpret several GP programs in parallel, thus obtaining speedups also for small training sets starting at less than 100 training cases. Our scheme was embedded in the well-known ECJ library, providing an easy entry point for owners of G80 GPUs.

1 Introduction

Newly introduced graphics processing units (GPUs) provide fast parallel hardware for a fraction of the cost of a traditional parallel system. GPUs are designed to efficiently compute graphics primitives in parallel in order to produce the pixels of the video screen. Driven by ever increasing requirements from the video game industry, GPUs have evolved into very powerful and flexible processors, while their price remained in the range of consumer market. They now offer floating-point calculation much faster than today's CPU and, beyond graphics applications, they are very well suited to address general problems that can be expressed as data-parallel computations (i.e. the same code is executed on many different data).

Moreover, several general purpose high level-languages for GPUs have become available such as Brook[1] and thus developers do not need any more to master the extra complexity of graphics programming APIs when they design non graphics applications[2]. In this paper, we work with an Nvidia GeForce 8800GTX graphics card that is built around the G80 GPU. We used an NVidia provided extension to the C language, named CUDA (Compute Unified Device Architecture) that runs on the G80 GPU family, allowing fine control over the hardware capabilities. Note that this toolkit is not available for other manufacturers hardware, and is not backward compatible with older Nvidia GPUs.

Up to now, exploiting the power of GPUs within the framework of evolutionary computation has been done mostly for genetic algorithms, e.g. [1,2,3,4,5]. At

[1] http://graphics.stanford.edu/projects/brookgpu/

[2] See `http://www.gpgpu.org` for a survey.

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 98–109, 2008.
© Springer-Verlag Berlin Heidelberg 2008

the time of the writing of this paper, using GPUs for Genetic Programming is still fresh matter: a first approach using Microsoft's Accelerator toolkit has been proposed by Harding and Banzhaf [6,7], a tutorial-like paper using a graphics API approach was issued by Chitty[8], and a technical report using the Rapid-Mind development kit has been made available by Langdon [9]. However we may expect a quickly increasing number of studies in the near future.

Harding and Banzhaf's and Chitty's works are both based on the same approach: every GP individual is compiled for the GPU native machine code and then evaluated on the fitness cases using the parallel ability of the GPU. This scheme is iterated on every individual, until the whole population has been evaluated. These authors have obtained interesting speedups but mainly for large individuals and/or several thousands fitness cases. In [6] this is acknowledged as a weakness of this scheme: "Many typical GP problems do not have large sets of fitness cases..." and "this leads to a gap between what we can realistically evaluate, and what we can evolve". We also think that evolving programs with hundreds of thousands training cases is not the most common setting in today's GP problems. For example, GP is often used to perform supervised classification, and it may be difficult to provide large sets of labeled training cases, noticeably when labeling requires human intervention like medical diagnosis. Moreover, a look at Koza's et al. last book (chapter 15 in [10]) suggests that solving real world problems with GP may be more in need of large populations (up to 5,000,000 individuals in [10]) than large data sets.

In order to also exploit the power of the GPU on training sets of modest size, we propose another parallelization scheme. Instead of evaluating sequentially the GP solutions, parallelizing the training cases, we share the parallel capacity of the GPU between GP programs and data. Thus we evaluate different GP programs in parallel, and assign to each of them a cluster of elementary processors to treat the training cases in parallel. This yields more data to fill the pipeline of each ALU of the GPU, in order to improve the efficiency. As a consequence more computational power is available for e.g. increasing the population size.

As a consequence we must emulate a MIMD task (running different programs) on a basically SIMD hardware. A solution to this problem has been proposed in the 1990s [11] in the form on an interpreter that considers the set of programs as data. This was implemented for GP in the late 90s by Juillé and Pollack [12] on the MASPAR machine, and a similar technique is also proposed by Langdon [9] on the G80 GPU. Our approach differs from Langdon's since we use the CUDA development kit that allows a finer grain access to the hardware. Thus we can exploit a characteristic of the G80: it runs in Single Program Multiple Data (SPMD) mode, rather than SIMD, i.e. elementary processors run the same program (the interpretor) in parallel but they are divided into clusters that share their own program counter. This gives the opportunity to achieve increased speedups, since e.g. a cluster can interpret the "if" branch of a test while another cluster treat independently the "else" branch. On the opposite, performing the same computation inside a cluster is also possible, but the two

branches are processed sequentially in order to respect the SIMD constraint: this is called divergence and of course it is less efficient.

We interfaced our CUDA based evaluation with the popular ECJ library[3], and retained the most part of its flexibility. Our experiments have been done with the mainstream tree representation for GP individuals, using tutorial benchmarks taken from the ECJ library. The GPU speedup values that we are giving are for complete evolution runs and not only for the evaluation phase. Thus we hope these figures are close to the speedup readers may expect with their usual setting. An archive containing a sample code for a regression application is available at http://www-lil.univ-littoral.fr/~robillia/EuroGP08/gpuregression.tgz .

The rest of the paper is organized in the following way: next section provides some information on the graphics processing unit and the CUDA programming language. In Section 3, the implementation of the GP system is described. Section 4 presents benchmarks and results. Section 5 concludes and discusses future works.

2 The G80 GPU Architecture Overview

The graphics card we used is a NVidia GeForce 8800 GTX based on the G80 GPU. It is natively limited to single precision floating point (32-bit data precision), although double precision can be used through a software library. This hardware is based on an unified architecture: instead of the traditional specialized vertex and fragment processors that are found on many graphics cards, here the elementary processors are identical and managed as a pool of 16 so-called multiprocessors. A multiprocessor contains 8 elementary scalar stream processors that operate at a 1.35 GHz clock rate, giving a total number of 128 elementary stream processors on the graphics card. A multiprocessor also owns 16 kb of fast memory that can be shared by its 8 stream processors. Multiprocessors are SIMD devices, meaning their 8 stream processors execute the same instruction at every time step on its own data. However alternative and loop structures can be programmed. Let us suppose we execute a *while* structure and the conditional expression results as false for only one of the 8 stream processors that are contained in a multiprocessor. Then this stream processor is simply put into idle mode during the remaining loops performed by the others. This is called divergence, and of course it implies some waste of computing power.

Due to its architecture, the G80 GPU is able to function in SPMD mode (Single Program, Multiple Data) at the level of the multiprocessors: every multiprocessor must run the same program, but each of them owns its private program counter, thus they do not need to execute the same instruction at the same time step (as opposed to their internal stream processors). This flexibility allows to avoid divergence between multiprocessors by carefully dispatching the tasks on them, but up to now it can only be accessed with the CUDA development kit proposed by NVidia. Other toolkits consider this GPU only as a SIMD device containing 128 elementary processors, thus increasing the risk of divergence and

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

wasting computing power. This is why we used CUDA when implementing our population parallel scheme.

Note that our machine was equipped with another graphics card dedicated to display the X screen, while the 8800GTX card was reserved for the computations and thus not attached to a X server. This dual cards setting allowed us to obtain cleaner timings (no interference with the display). Note that it is perfectly possible to use the 8800GTX for both display and GP evolution, with some constraints: during intensive computation, the user interaction with the X desktop is suspended; moreover any given call to the GPU (i.e. executing the interpreter in our case) cannot last more than 5 seconds, otherwise the process is killed by the X server.

3 Population Parallel Model

As said above, previous works about GP on GPU have demonstrated interesting speedups for very large training sets and/or programs. Indeed if we execute one GP program at a time on the G80, parallelizing only the training data as it is proposed in [6], then we do not have enough data to fill all ALU pipelines of the 128 stream processors, thus the GPU is under-exploited. This phenomenon, in addition to compilation overheads, may also explain the bad GPU timings observed by [6] on the 7300 GPU with small training cases sets. A possible solution is to evaluate several programs in parallel to increase the computation load.

As the G80 is a SPMD device, SP meaning *Single Program*, we cannot perform the direct execution of several *different* programs in parallel. The same problem arose for Juillé and Pollack when they implemented GP on the MASPAR machine [12] and they proposed to bypass this limitation by interpreting GP solutions. In the same way, we run one program on the GPU: an interpreter dedicated to execute any GP program for our benchmark problems. The GP programs are simply considered as data from the interpreter point of view. This is clearly a trade-off choice: the computing time of iterating interpreted code on training cases is to be balanced against the time of compiling and iterating a compiled code. Few training cases means few iterations thus the interpreter may be a sensible trade-off.

In order to interpret the GP programs, we first have to copy them into the graphics card memory. This is not straightforward, since we want to integrate the GPU evaluation inside the ECJ library, while retaining the most part of its flexibility. Indeed, GP tree nodes in ECJ are scattered into memory, so we need to compact them into a single chunk of memory that can be transfered to the GPU. We also translate the trees into linear stack-based postfixed notation code that will be easier to interpret, although it is not required. This is illustrated in Figure 1.

The interpreter code is run on the GPU and is quite simple, being composed of a main loop fetching the next instruction to process, and a switch that performs the operations required depending on this instruction, see pseudo-code in Table 1. The *if* structure and the short-circuit $\{And, Or\}$ operators are implemented as usual in code generation, i.e. bypassing branches that do not need to

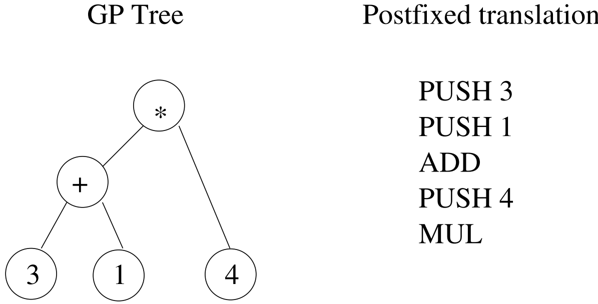


Fig. 1. Sample GP tree and its translation into linear postfixed notation, prior to its interpretation

Table 1. Pseudo code of the interpreter

```
sp = 0 ; // initialize stack pointer
GP_pc = baseAddress[i] ; // load base address of prog i
while (instructionArray[GP_pc] != RETURN) {
   switch (instructionArray[GP_pc]) {
      case OPERAND : stack[sp++] = data; // push data on stack
      case ADD : stack[sp-2] = stack[sp-1] + stack[sp-2]; sp--;
      case MUL : stack[sp-2] = stack[sp-1] * stack[sp-2]; sp--;
      ...
   }
   GP_pc++;
}
```

be evaluated (see [13]). A detailed GP oriented implementation is found in [12]. We use a stack to hold temporary results.

In order to limit the occurrences of divergence, we dispatch the population of GP trees in such a way that, at any time, each multiprocessor interprets only one GP tree. That is, GP trees are parallelized on the multiprocessors, giving up to 16 GP programs interpreted in parallel on the G80, and the fitness of a given tree is in turn parallelized on the 8 stream processors contained in the multiprocessor. This scheme is illustrated in Figure 2. So every stream processor evaluates $1/8^{th}$ of the training cases. This $1/8^{th}$ factor leaves enough data to fill the ALU pipelines in most cases, even with small training sets. In a scheme where only one GP program is run and only the training data are parallelized, each stream processor receives only $1/128^{th}$ of the training cases and this leads to under-exploitation with small training sets.

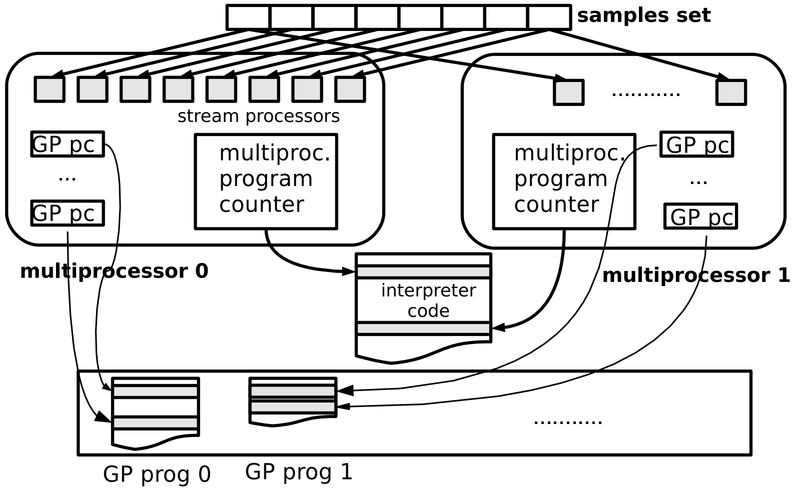


Fig. 2. Parallelization scheme: multiprocessors independently execute the interpreter code. On every multiprocessor, each stream processor handles a part of the training set and stores in register memory the current address of the GP program instruction to be interpreted (GP pc). These GP pc do not need to point to the same instruction of their GP program. However, if the instructions to be interpreted in parallel are not the same for all the stream processors on a given multiprocessor, this will imply divergence and loss of efficiency while some stream processors wait in idle mode.

To sum up some characteristics of our scheme:

- when the evaluation of a GP program is finished on a multiprocessor, there is no need to wait in idle mode for the completion of programs that are interpreted on other multiprocessors: another GP tree can be interpreted immediately; this is possible because we work in SPMD mode, versus the SIMD scheme proposed by [12,9];
- the same holds when two different programs contain *if* or *loop* instructions: this does not create divergence between programs;
- however we can incur divergence between stream processors on the same multiprocessor, as they always work in SIMD mode, when e.g. an *if* structure resolves into different cases within the set of 8 training cases that are processed in parallel[4].

4 Results and Discussion

In this section, we assess the performance of our parallelization scheme on the G80 GPU against an Intel 2,6 GHz CPU (single core). We used three standard

[4] Actually this is a bit more complex since CUDA schedules multiples of 4 computations per stream processor in order to amortize memory access overheads. A detailed explanation is not possible within the size constraints of this paper.

benchmarks taken from the ECJ library: real and boolean regression and a classification problem. Two of these benchmarks were also used by [6], although it is not possible to perform a direct comparison since we do not use the same hardware. Anyway, we do not focus on GP being able to solve these standard benchmarks — this has been covered in numerous previous works — but rather on the computing time speedup that can be brought by the GPU. Timings are monitored for the evaluation phase, that includes translation to postfixed code in the GPU case, and also for full evolutionary runs.

All runs were done using 32-bits floating point arithmetics on both CPU and GPU. We noticed small differences between the fitness values computed on both schemes. These differences were about 10^{-7} in magnitude and are implied by the parallelization scheme: the raw fitness is cumulated into a single loop on the CPU, while on the GPU each stream processor computes the fitness associated to its part of the training cases before the global result is cumulated. Thus small rounding errors appear that can change the result of the evolutionary selection phase, especially with the bigger populations where probabilities are higher to meet individuals with very close fitness values. These rounding errors tend to accumulate over generations and can yield slightly different runs between CPU and GPU. Here the situation is somewhat comparable to what happens when one performs a GP benchmark between machines with different precision levels. Thus, in order to obtain significant figures, we have done 30 independent evolutionary runs for each problem and setting, then we have averaged the running times. In turn, these average times are divided by the average evolved tree size observed respectively for the CPU and the GPU, in order to obtain a comparable time per node ratio. The speedup indicates how many times the GPU version is faster than the CPU one and is computed as:

$$\text{speedup} = \frac{\text{GPU mean tree size}}{\text{CPU mean tree size}} * \frac{\text{CPU mean running time}}{\text{GPU mean running time}}$$

The first benchmark is the standard regression problem $x^6 - 2x^4 + x^2$ (see [14]), using population sizes of 100, 500, 2500 and 12500 individuals, 50 generations, and training set sizes of 64, 256 and 1024 training cases. The function set is {+, -, *, /, sin, cos, exp, log} and terminal set {ERC (i.e. Ephemeral Random Constants), X}. Depending on the population and training set sizes, the average evolved tree size ranges from 30 to 66 nodes. Speedup figures are shown in Figure 3.

The second benchmark is based on the multiplexer-6 and multiplexer-11 problems (see [14]) with respectively 64 and 2048 training cases, for population sizes 100, 500, 2500 and 12500 individuals, and 50 generations. We used as function set {And, Or, Not, If} and terminal set {A0-A1, D0-D4}, resp. {A0-A2, D0-D7}. The "And" , "Or" and "If" are shortcut versions (i.e. bypassing branches that do not need to be evaluated) and boolean values are stored as integers, to obtain comparable results with the ECJ standard code. Depending on the population and training set sizes, the average evolved tree size ranges from 112 to 157 nodes. Speedup is illustrated in Figure 4.

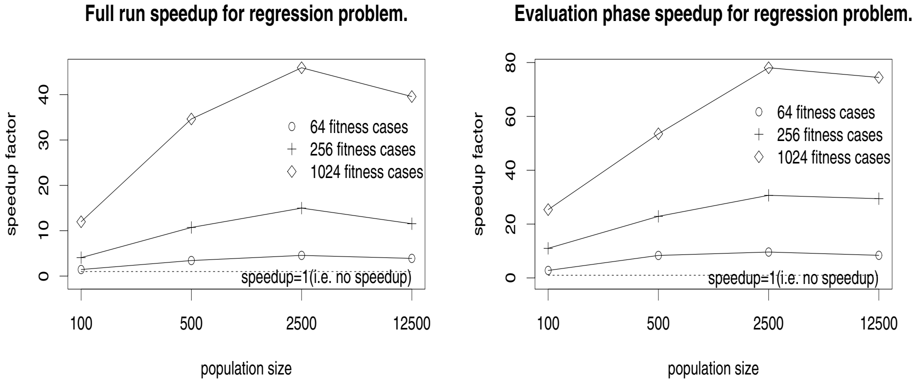


Fig. 3. GPU vs CPU speedup on regression problem $x^6 - 2x^4 + x^2$. On the left, speedup for whole evolutionary runs, on the right speedup for evaluation phase only.

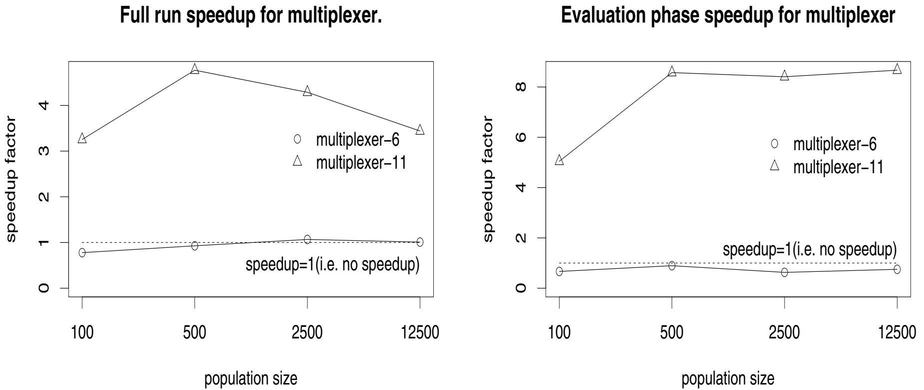


Fig. 4. GPU vs CPU speedup on multiplexer-6 and multiplexer-11 (64 and 2048 training cases respectively). On the left, speedup for whole evolutionary runs, on the right speedup for evaluation phase only.

The third benchmark is the intertwined spirals problem (see [15]), again population sizes range from 100 to 12500 individuals, and the training set size is fixed to 194. The function set is {+, -, *, /, sin, cos, Iflte[5]} and the terminal set is {ERC, X}. Depending on the population size, the average evolved tree size ranges from 119 to 208 nodes. Speedup is illustrated in Figure 5.

These Figures show that, in all but one cases, evaluation on the GPU yields a speedup in computing time, for small training cases sets and short expression lengths: the largest average tree size we encountered was 208 nodes. However the CPU is superior for the multiplexer-6 problem: the memory transfer and

[5] *Iflte* is a quaternary operator that stands for "If *sibling1* less than *sibling2* then *sibling3* else *sibling4*.

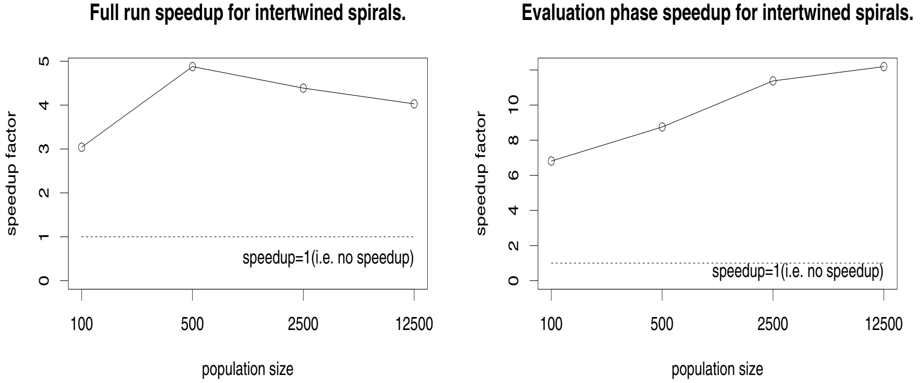


Fig. 5. GPU vs CPU speedup on intertwined spirals (194 training cases). On the left, speedup for whole evolutionary runs, on the right speedup for evaluation phase only.

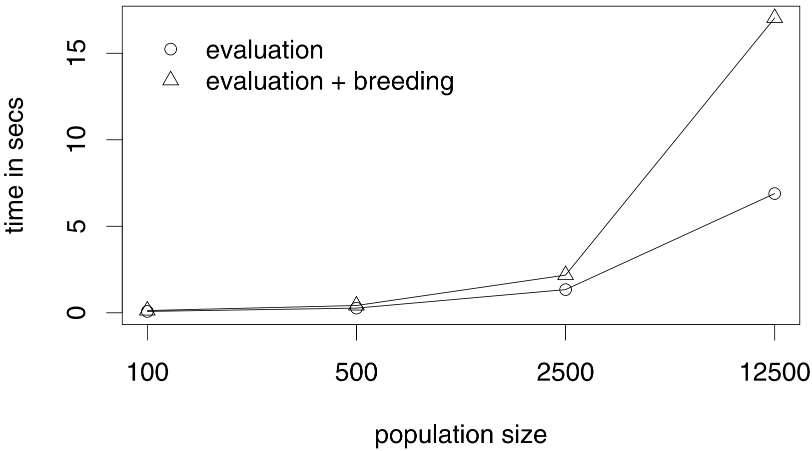


Fig. 6. Mean evaluation and breeding time for the GPU runs on the $x^6 - 2x^4 + x^2$ regression problem, with 1024 cases. As breeding is kept on the CPU, it becomes the bottleneck when processing large populations.

tree-to-postfix translation overheads cannot be counter-balanced by the speedup in parallel computation. Note that typical solutions to this problem contain many *If* operators and may be suspected to create a high divergence between the 8 inner stream processors that deal with one GP program. This means that many branches of the GP tree must be interpreted with part of the stream processors in idle mode to respect the SIMD constraint, with a drop of performance.

For full runs, the speedup increases with the population size until we reach a threshold where it begins to stagnate or drop. Of course the speedup cannot

increase indefinitely and must anyway reach an upper bound when the GPU is saturated. But in our case, this phenomenon occurs earlier due to one basic implementation choice: the breeding phase is done by the ECJ library, so it is computed on the CPU and its cost increases faster than the evaluation cost, as can be seen on Figure 6 for the regression problem $x^6 - 2x^4 + x^2$. We recall that the evaluation time includes the cost of compacting the programs in linear form and translating them to postfixed notation. This phase is also performed on the CPU in our current implementation, and thus does not benefit from parallelization. This is responsible for the slight drop of evaluation speedup in the right plot of Figure 3 with population size 12500.

At last, speedup factors obviously depend on the problem, especially if it needs operators such as “If” that create unavoidable divergence between stream processors, wasting computation cycles. This explains the difference in performance between the regression benchmark and the two others.

5 Conclusions and Future Works

Previous works about parallelizing GP on GPU brought speedups with respect to large programs or/and large training sets. However it is not always possible to gather large training sets, e.g. when labeling training cases requires human intervention like medical diagnosis. It is neither always easy to evolve large GP individuals without incurring a high level of over-fitting. Thus it is also interesting to obtain speedups for small GP trees and small training cases.

We worked on one of the current fastest GPU, the Nvidia G80. Our solution consists in parallelizing both GP programs and training data, as opposed to run sequentially each compiled program and parallelizing only the training set. When several programs are run in parallel, they process proportionally more training cases on each elementary processor of the GPU, so we can expect a better filling of the ALUs and an overall increased efficiency. As running different GP programs on a basically SIMD architecture is not possible, we use an interpreter to process both programs and training cases as data. On a SIMD device, a typical problem raises when the main interpreter switch is required to execute different instructions in parallel: the GPU executes these instructions sequentially, putting alternatively some of the elementary processors in idle mode. This is called divergence. As our scheme relies on a fine grain parallelization allowed by the CUDA language, we have the opportunity to exploit the SPMD architecture of the G80, i.e. this GPU is composed of a set of 16 independent SIMD multiprocessors. We dispatch one program per multiprocessor, thus divergence appears only in the case when the GP function set contains “if” or “loop” nodes.

With this parallelization scheme we obtained evaluation phase speedups ranging from 8 times to 80 times for 5 out of 6 benchmarks, using from 64 to 1024 training cases and mean evolved tree sizes from 30 to 208 nodes. However no speedup was obtained on the multiplexer-6 benchmark, which cumulates a small training set together with a high tendency to create divergence through the major part of its function set (if, shortcut and, shortcut or).

By implementing the GPU evaluation as part of the Java ECJ library, we also allow other users of G80 cards to easily develop their own GP applications. However, keeping ECJ flexibility has a drawback: the breeding phase is performed on the CPU and does not benefit from the GPU power. Experiences showed that when population size increases, the breeding time grows until it is no more negligible against the evaluation time.

Future works include extending ECJ to store the GP population into an array and implementing an interpreter for prefix code, removing the need for compacting and translating GP trees to postfix. Nonetheless the cost of the breeding phase suggests that it is also required to implement it on the GPU, in order to take full advantage of the new graphics cards power to evolve large populations.

Acknowledgements. This work was partially supported by European Union Interreg IIIA project 182b.

References

1. 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), 9 April, vol. 3, pp. 2286–2293. IEEE, Los Alamitos (2005)
2. Yu, Q., Chen, C., Pan, Z.: Parallel genetic algorithms on programmable graphics hardware. In: Downey, R.G., Fellows, M.R., Dehne, F. (eds.) IWPEC 2004. LNCS, vol. 3162, pp. 1051–1059. Springer, Heidelberg (2004)
3. Kaul, K., Bohn, C.-A.: A genetic texture packing algorithm on a graphical processing unit. In: Proceedings of the 9th International Conference on Computer Graphics and Artificial Intelligence (2006)
4. Wong, T.-T., Wong, M.L.: Parallel Evolutionary Computations. In: chapter 7, pp. 133–154. Springer, Heidelberg (2006)
5. Fok, K.-L., Wong, T.-T., Wong, M.-L.: Evolutionary computing on consumer graphics hardware. In: IEEE Intelligent Systems, pp. 69–78 (2007)
6. Harding, S., Banzhaf, W.: Fast genetic programming on GPUs. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 90–101. Springer, Heidelberg (2007)
7. Harding, S., Banzhaf, W.: Fast genetic programming and artificial developmental systems on GPUs. In: proceedings of the 2007 High Performance Computing and Simulation (HPCS 2007) Conference, p. 2. IEEE Computer Society, Los Alamitos (2007)
8. Chitty, D.M.: A data parallel approach to genetic programming using programmable graphics hardware. In: Proceedings of the 2007 Genetic and Evolutionary Computing Conference (GECCO 2007), pp. 1566–1573. ACM Press, New York (2007)
9. Langdon, W.B.: A SIMD interpreter for genetic programming on GPU graphics cards. Technical Report CSM-470, Department of Computer Science, University of Essex, Colchester, UK, 3 (July 2007)
10. Koza, J., Keane, M., Streeter, M., Mydlowec, W., Yu, J., Lanza, G.: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers, Dordrecht (2003)

11. Sanders, P.: Emulating mimd behavior on simd machines. In: Proceedings of International Conference on Massively Parallel Processing Applications and Development, Elsevier, Amsterdam (1994)
12. Juille, H., Pollack, J.B.: Massively parallel genetic programming. In: Advances in Genetic Programming 2, vol. 17, pp. 339–358. MIT Press, Cambridge (1996)
13. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers — Principles, Techniques and Tools. Addison-Wesley, Reading (1986)
14. Koza, J.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
15. Lang, K.J., Witbrock, M.J.: Learning to tell two spirals apart. In: Morgan-Kaufmann (ed.) Proceedings of the 1988 Connectionist Summer Schools (1988)

Operator Equalisation and Bloat Free GP

Stephen Dignum and Riccardo Poli

Department of Computing and Electronic Systems,
University of Essex,
Wivenhoe Park, Colchester, CO4 3SQ, UK
{sandig,rpoli}@essex.ac.uk

Abstract. Research has shown that beyond a certain minimum program length the distributions of program functionality and fitness converge to a limit. Before that limit, however, there may be program-length classes with a higher or lower average fitness than that achieved beyond the limit. Ideally, therefore, GP search should be limited to program lengths that are within the limit and that can achieve optimum fitness. This has the dual benefits of providing the simplest/smallest solutions and preventing GP bloat thus shortening run times. Here we introduce a novel and simple technique, which we call *Operator Equalisation*, to control how GP will sample certain length classes. This allows us to finely and freely bias the search towards shorter or longer programs and also to search specific length classes during a GP run. This gives the user total control on the program length distribution, thereby completely freeing GP from bloat. Results show that we can automatically identify potentially optimal solution length classes quickly using small samples and that, for particular classes of problems, simple length biases can significantly improve the best fitness found during a GP run.

Keywords: Genetic Programming, Search, Bloat, Program Length, Operator Equalisation.

1 Introduction

An intrinsic feature of traditional Genetic Programming (GP) is its variable length representation. Although, this can be considered one of the method's strengths, researchers have struggled with the phenomenon of bloat, the growth of program size during a GP run without a significant return in terms of program fitness, since GP's inception.

Numerous theories to explain bloat have been put forward including Replication Accuracy [1], Removal Bias [2], Nature of Program Search Spaces [3] and, more recently, Crossover Bias [4]. Numerous methods to control bloat have also been suggested [3,5,6], including, for example, size fair crossover or size fair mutation [7,8], Tarpeian bloat control [9], parsimony pressure [10,11,12], or using many runs each lasting only a few generations.

Research has shown that beyond a certain minimum program length the distributions of program functionality and fitness converge to a limit [13]. Before that limit, however, there may be program-length classes with a higher or lower average fitness than that achieved beyond the limit. Ideally, therefore, GP search

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 110–121, 2008.
© Springer-Verlag Berlin Heidelberg 2008

should be limited to program lengths that are within the limit and that can achieve optimum fitness. We might want, for example, to restrict our search fixing program sizes at the point where our smallest optimal or near optimal solutions can be found thereby avoiding the need to search much larger spaces with the additional computational effort that entails. For most applications simpler solutions are also much more desirable than larger solutions.

In this paper we provide a method, *operator equalisation*, that can be applied easily to existing GP systems. This method forces GP to search specific length classes using pre-determined frequencies so that we can control the sampling rates of specific program lengths. As explained in Section 2, the method is very simple. This technique has several advantages. For example, whenever the length distribution and the corresponding sampling bias provided by standard operators is not suitable for a specific program space, we can change such a bias freely making it possible to sample or oversample certain length classes we believe can benefit our search. The user is given complete control over the program length distribution, and bloat can be entirely and naturally suppressed by simply asking operator equalisation to produce a static length distribution. We look at how different, static target length distributions can influence performance in Section 4. Furthermore, this method enables us to automatically sample and exploit the best fitness values associated with particular length classes as explained in Section 5.

2 Operator Equalisation

Investigations into the properties of program length have often used the tool of histogram representation in order to compare frequencies of programs sampled at particular lengths during a GP run [14,15,4]. Our operator equalisation method aims at controlling the shape of length histograms during a run. The method is loosely inspired by both the gray-level histogram equalisation method [16] used in image processing and digital photography to correct underexposed or overexposed pictures and the Tarpeian bloat control method [9] which, with a certain probability, by setting to zero the fitness of newly created programs of above average length effectively suppresses their insertion in the population. We have taken these ideas forward to see if by filtering which programs are allowed to be inserted in the population we can directly manipulate those frequencies in order to force GP to sample programs of particular lengths at pre-specified rates.

The method requires users to specify the desired length distribution (which we will call `target`) that they wish the GP system to first achieve and then continue to use when sampling a program space. This allows one to specify both simple well known probability distributions (Section 4) and also coarser grained models (Section 5). During the initialisation of the GP system a `histogram` object is created. This needs only to be primed with the maximum size allowed, number of bins (the size of the bins being calculated from these) and of course the `target` distribution. Then the method requires wrapping the existing code for offspring generation with code that simply accepts or disallows the creation of a child based on its length. The wrapper is extremely simple:

```
repeat {
  <create a child using standard genetic operators>
} until( histogram.acceptLength( child.length ) )
```

Internally, the histogram object maintains a set of numbers, one for each length class, which act as acceptance probabilities. The acceptLength method simply generates a uniform random number between 0 and 1 and compares it against the acceptance probability associated with the length class associated to child, returning true if the random number is less than the acceptance probability, and false otherwise.

At the end of each generation the histogram object updates the acceptance probabilities for each class using the following formula:

```
newProbability = currentProbability + ( normalisedDiff * rate )
```

where normalisedDiff is the difference between the desired frequency specified in target and the current frequency divided by the desired frequency. Small discrepancies for large classes are, therefore, largely ignored. The user defined parameter rate determines how quickly the distributions should converge. After some experimentation the setting rate=0.1 was found to work well and has been used in all experimentation presented in this paper.

As one can see the method can easily be applied to existing GP applications with minimal changes: users need to change only very few lines of code in their existing GP systems.

3 Test Problems

We have deliberately chosen two GP problems of differing natures, a parity problem and a symbolic regression problem, to show the benefits and limitations of this approach. As we will show the first requires a relatively large program size before fitness will significantly improve whilst the second is able to achieve relatively high, though far from optimal, fitness values with small program sizes.

The Even Parity problem attempts to build a function that evaluates to 1 if an even number of boolean inputs provided evaluate to 1, 0 otherwise. We have chosen a relatively large input set of size 10. However, it is possible to evaluate all possible fitness cases (1024) for each potential solution within a reasonable time given the short length limit imposed.

Our second problem is a 10-variate symbolic regression problem: $x_1x_2+x_3x_4+x_5x_6+x_1x_7x_9+x_3x_6x_{10}$ as described in [9][1], which we have called Poly-10. 500 test cases are used each comprising of a (uniform) randomly generated value for each variable ranging between -1 and 1 and the resulting value of the equation.

As with the Even-10 problem only functions with arity 2 are used: ADD, SUBTRACT, MULTIPLY and a protected division function called PDIV which returns the denominator if the resulting division is less than 0.001. No Ephemeral Random Constants (ERCs) were used.

[1] This problem can be simplified to $x_1(x_2+(x_7x_9))+x_3(x_4+(x_6x_{10}))+x_5x_6$ to give a smallest GP tree size of 19 nodes.

Both problems have been sourced from [9] with minor alterations[2] to enable comparison and analysis. Each problem is expected to bloat under non-constrained conditions the reasons for which are described in the original paper.

4 Equalising to Simple Program Length Distributions

All experiments were initialised using the GROW method [17] with depth 6. For simplicity subtree swapping crossover with uniform selection of crossover points was applied without mutation or replication. Elitism was not applied. We used tournament selection with tournament size of 2 in experimentation. The algorithm was generational. All experiments used a population of 10,000 and ran for 100 generations. Results were averaged over 100 runs. It should be noted that due to the wrapper-like implementation there is no reason why mutation, replication or other forms of crossover could not be applied in isolation or combination. In fact it is hard to imagine any form of standard GP experimental set-up which could not be used easily.

In order to satisfy our stated desire of bloat free GP we have chosen a strict, deliberately small, length limit of 100 nodes. This has the added benefit of allowing us to evaluate a large set of fitness cases for each potential solution within acceptable experimental run times.

4.1 Does Operator Equalisation Work?

Initial investigation using our parity and regression problems showed that using a fairly unforgiving initialisation method (GROW), i.e., that in no way matched to our desired length distributions, we could equalise program lengths within approximately 20 generations. This is shown in Figure 1 for the Poly-10 problem equalised for a uniform length distribution.

With both problems there was a small dip for some of the early length classes. This is due to the fact that when the population has a uniform length distribution, crossover is less likely to produce very short programs than is ordinarily the case in the absence of equalisation. This is illustrated in Figure 2 where we look at the number of programs rejected by the wrapper at generation 100. As we can see the number of programs rejected for these length classes is extremely small. Our equaliser is, therefore, doing the best with what it has been presented by the underlying GP system.[3] The smallest class was always well populated. As the bias of subtree crossover towards sampling programs of a single node has been widely reported in the literature this is of little surprise.

[2] Our Even-10 problem has no NOT function. So all functions have an arity of 2. Also, Poly-10 here uses 500 fitness cases, where originally 50 were used.

[3] In other experiments (not reported) we found that the dip is slightly worsened by the use of larger tournament sizes since this increases the ability of selection to repeatedly present certain program sizes. The effect is, by contrast, reduced when using a steady state model, as GP can select newly created programs, i.e., those accepted by our equaliser, immediately without having to wait for a generation to be completed.

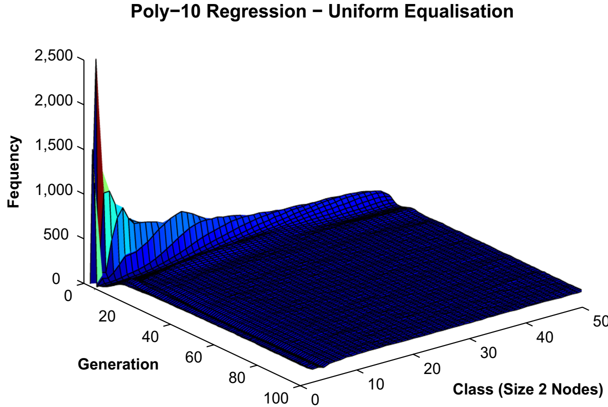


Fig. 1. Length histogram for Poly-10 regression problem with uniform equalisation of program length classes

Although it is possible to imagine extreme conditions where infinite loops could be encountered, for the experimentation detailed in the following sections, all runs were completed succesfully and no unusually large run-times were recorded. It is of course possible to add a simple retry limit to the wrapper code to escape such loops.

4.2 Efficiency of Different Length Distributions

Having established that our simple operator equalisation algorithm works for our test problem, we then applied this method to see how the use of elementary, easily recognisable, target probability distributions could affect our search. In this paper, we only consider static distributions, although operator equalisation works also with dynamic targets—a case that we will study in detail in future research.

In Figure 3 we see the final length distributions for the parity problem, i.e., at generation 100, for different target distributions. Each length class is 2 nodes in size. Given that all the functions in our function set (AND, OR, NOR, NAND, XOR and EQ) have an arity of 2, we have an individual class for every possible length up to our size limit.

We have chosen to look at a uniform distribution where each length is sampled with the same frequency, a triangular distribution which has a linearly increasing bias towards sampling larger programs, a 'reverse' triangle where smaller programs are sampled more often and a reverse exponential distribution where we sample larger programs exponentially more frequently than shorter ones. Note, the distribution for the length limit with no equalisation is also shown. In all cases the target distribution was reached very quickly. For example, after some initial fluctuations, as we can see in Figure 4, the average size for each of the experiments settled to a fixed value.

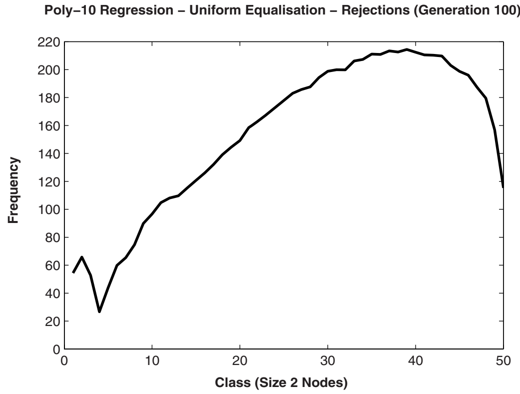


Fig. 2. Number of equaliser rejections at generation 100 for Poly-10 regression problem with uniform equalisation of program length classes

If we compare the best fitness values recorded for different target distributions (Figure 5), we can see that the push towards sampling larger programs has had a beneficial effect compared to using the simple length limit. The exponential distribution has a sharper upwards gradient for generations 20 to 60 than that of the triangular distribution although both eventually converge to the same value. The bias towards the sampling of smaller programs has had the most negative effect. Selection does, however, manage to improve fitness in all of our experiments. Perhaps surprisingly, all equalisation methods improve the best fitness value compared to the simple length limit during the early generations. The value of exploring certain length classes during early generations is discussed further below.

Unlike the Even-10 problem we can see in Figure 6 that the imposition of target length distributions has a negative effect on all forms of equalisation for best fitness compared to our simple length limit for the Poly-10 problem, any undersampling of smaller programs during the early generations having the most marked effect. It has long been known that in symbolic regression problems smaller programs can obtain relatively high fitness. In fact, the reverse triangle distribution performs as well as the simple length limit up to generation 15 and outperforms most other methods most of the time. This indicates that in this problem the dynamics of the length distribution is important, and GP benefits from exploring short programs for 10 or 15 generations and then progressively moves towards sampling longer programs, as GP with a simple length threshold does. So, this suggests that there could be benefits in using dynamic target distributions. As previously mentioned, we will explore this in future work.

Methods to detect this bias are discussed in the next section.

5 Length Class Sampling

As we have a method to directly influence the sampling of particular length classes, we can now look at two sampling techniques that can help us gain an

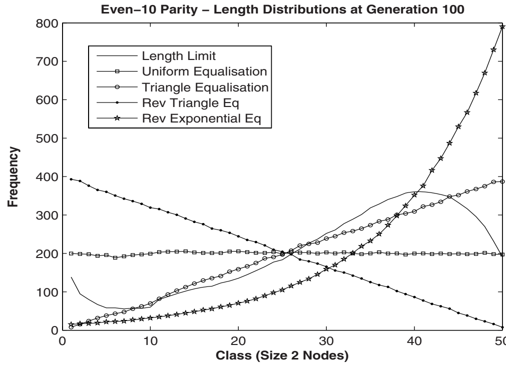


Fig. 3. Final length distributions for Even-10 parity problem using a strict length limit and different equalisation targets

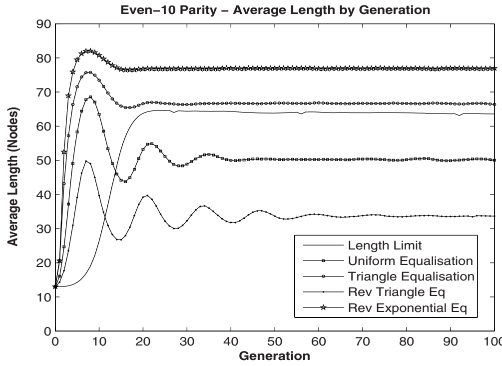


Fig. 4. Average length (nodes) for Even-10 parity problem using a strict length limit and different equalisation targets

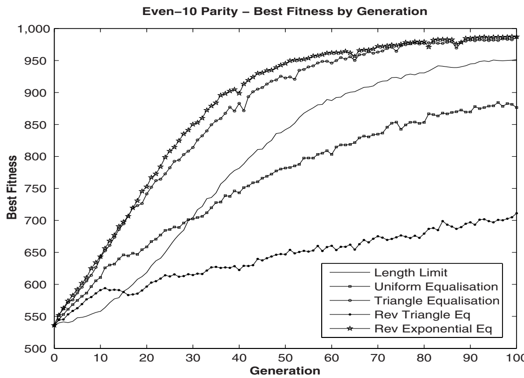


Fig. 5. Best fitness (number of test cases matched) for Even-10 parity problem using a strict length limit and different equalisation targets

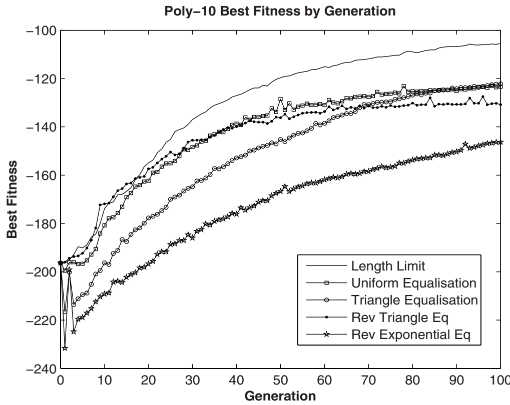


Fig. 6. Best fitness (Minus Mean Squared Error) for Poly-10 Symbolic Regression problem using a strict length limit and different equalisation methods

insight into the program space that we wish to search. We present these techniques in Sections 5.1 and 5.2. Both problems from the previous section were investigated using them.

Note that for the experiments described below we used the same GP system as in Section 4, but with two small, yet important, differences. Firstly, in order to remove any initial length bias the GROW initialisation method has been replaced with the RAND_TREE method described in [18]. Secondly, to show that useful insights into the program space of a problem can be achieved without undue computer resources, we have used both a smaller length limit of 80 nodes and a much reduced population size of 1,000.[4]

5.1 Single Length Classes

Using the RAND_TREE method we can sample without bias specific length classes. We can, therefore, look at the sampling of individual classes in isolation. For our experimentation the search space was divided into 20 equal length classes with each class sampling two distinct program lengths e.g. 1 and 3 for the first class, 5 and 7 for the second etc.[5] The objective was to find out which area (length class) of the search space would appear preferable to a GP system in the early generations of a run.

For the Even-10 problem (Figure 7) we can see quite clearly that there is small threshold where potential solutions cannot achieve anything better than 512 correct classifications, exactly half the total possible. However, as we move to larger program sizes we can see a distinct improvement in fitness. Selection will, therefore, quickly guide GP to larger programs in the early stages of a GP run.

[4] As we have used a smaller population size we cannot directly compare the best fitness results reported in this section with those reported in the previous section.

[5] Even sized programs are not possible for 2-ary trees.

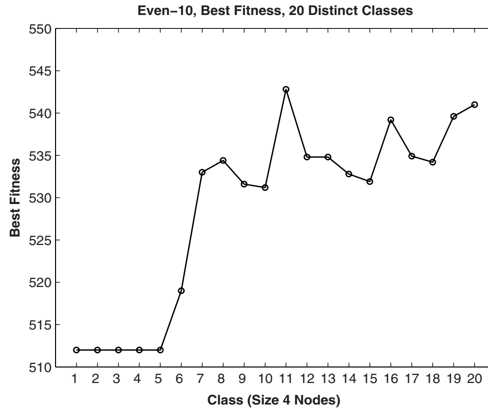


Fig. 7. Best fitness for Even-10 problem sampling 20 distinct size classes using the RAND_TREE method. Results are averages over 100 samples of 1,000 individuals each (1,000=GP population size).

Figure 8 shows that, for the Poly-10 problem, when we initially sample the program space, we find that the smallest programs do indeed have relatively better fitness than their larger counterparts. This explains GP's concentration in this area during earlier generations in the experimentation reported in Section 4. Of course, these areas do not contain optimal solutions: we need at least 19 nodes to achieve that. However, to an initial random sampling these areas display a higher proportion of relatively fit programs than those of the larger program size search spaces sampled. This explains why without histogram equalisation GP first samples the short programs but then quickly moves towards the longer programs, where, upon sufficiently sampling, better solutions can be found. This also explains why equalisation with a reverse triangular distribution does well initially, but cannot compete with standard GP later on (see Figure 6). Finally, it also explains why equalisation with distributions that sample the longer programs more frequently, such as the reverse exponential distribution, produce much worse fitness that standard GP and reverse triangular equalisation, initially.[6]

5.2 Multiple Length Classes

Of course the picture may change significantly if we sample two or more classes, perhaps with differing proportions. Also, what may look like a good sampling histogram initially (upon the random sampling produced by initialisation) may later turn out to be suboptimal after many generations of GP exploration. So, in this section we look at how the picture changes when using multiple length classes in combination and when comparing the initial to the final generation of runs.

[6] The fitness plot for the reverse exponential distribution in Figure 6, however, remains parallel to the plot of standard GP, suggesting that given enough generations histogram equalisation with this distribution would eventually catch up.

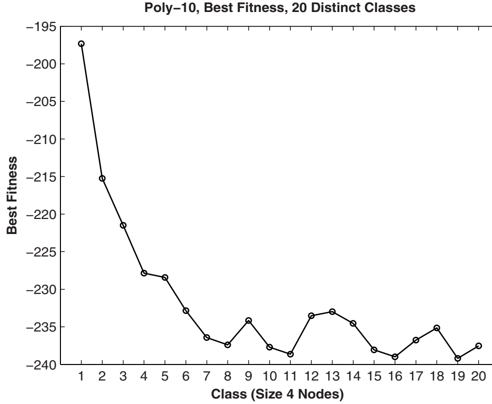


Fig. 8. Best fitness for Poly-10 problem in the same conditions as for Figure 7

To this end, we divided the length distribution into 4 bins of size 20 nodes and sampled each combination of bins using frequencies that were multiples of 20%. For example, bins 2 and 3 might have frequencies of 40% each, while bin 1 might have a frequency of 20% and bin 4 a frequency of 0%. Every combination (including those where some bins were empty) was sampled. There were 56 combinations in total. For each the resulting best fitness values at each generation were tabulated.

This produced a very large dataset, which we cannot report here due to space limitations. However, we report summaries of it in the form of the multiple linear regression formulas resulting from fitting the data at generations 0 and 100. The formulas will have the following form:

$$bestFitness = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_4 \quad (1)$$

β_0 is the constant term and β_i being the coefficient of each of the length classes X_i, X_1 being the smallest class. After the multiple linear regression was applied to the Even-10 problem the following formula was found for our initial generation:

$$bestFitness = 424.897 + 96.256X_1 + 103.899X_2 + 110.831X_3 + 113.911X_4 \quad (2)$$

As we can see there is a small improvement in best fitness as we search the larger classes. After applying GP search, at generation 100 the improvement is more distinct as shown by the regression formula:

$$bestFitness = 509.914 - 36.000X_1 - 12.000X_2 + 216.276X_3 + 342.748X_4 \quad (3)$$

For the Poly-10 problem for the first generation we obtain:

$$bestFitness = -179.595 - 15.509X_1 - 54.645X_2 - 56.678X_3 - 52.763X_4 \quad (4)$$

while after 100 generations the picture is somewhat different:

$$bestFitness = -147.146 - 33.712X_1 - 25.438X_2 - 46.835X_3 - 41.161X_4 \quad (5)$$

We can clearly see that for Poly-10 different parts of the search space yield different results for our initial generation and later stages of GP search. As one would expect from these results the best and worst combinations for our 100th generation showed a strong dislike for the third class. A 100% sampling of which, was indeed our worst result of -215.265, whilst more interestingly a broader sampling of the surrounding classes yielded the best results all of which were below -170.

6 Conclusions

In this paper we have introduced operator equalisation, a programatically simple method that can be easily applied to current experimental environments that allows us to finely bias GP search to specific program lengths. In particular, when method can force GP to sample the search space using static (and arbitrary) length distributions. This completely and naturally suppresses bloat.

We have applied this method to first see how simple bias can influence the results of two different but potentially bloating problems. The Even-10 parity problem was shown to have a simple positive bias towards longer programs within the 'experimentally-friendly' 100 node limit we have specified, whilst the Poly-10 regression problem was shown to have a positive bias towards the sampling of shorter programs during early generations.

Using simple statistical techniques we have then shown how we can use the method to quickly gain information about the search space and the best way to sample it with GP (with and without equalisation).

The primary aim of bloat free GP is to sample program spaces in such a way that we allow GP to discover optimal or acceptable near-optimal solutions without wasting resourses searching ever larger spaces with little return with regard to fitness. Here we have made some strong steps in this direction. An automatic method of defining the appropriate search space for a GP problem may not be so far off. For example, there is no reason why the method introduced in this paper cannot be applied to the initial setting of size limits (either maximum or minimum), or even to define a dynamic schedule for biasing the sampling of programs to certain sizes over the entire run or during different stages of a GP run.

References

1. McPhee, N.F., Miller, J.D.: Accurate replication in genetic programming. In: Eshelman, L. (ed.) Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA 1995), 15-19 July, pp. 303–309. Morgan Kaufmann, San Francisco (1995)
2. Soule, T., Foster, J.A.: Removal bias: a new cause of code growth in tree based evolutionary programming. In: 1998 IEEE International Conference on Evolutionary Computation, 5-9 May, pp. 781–786. IEEE Press, Los Alamitos (1998)
3. Langdon, W.B., Soule, T., Poli, R., Foster, J.A.: The evolution of size and shape. In: Spector, L., Langdon, W.B., O'Reilly, U.-M., Angeline, P.J. (eds.) Advances in Genetic Programming 3, vol. 8, pp. 163–190. MIT Press, Cambridge (1999)

4. Dignum, S., Poli, R.: Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In: Thierens, D., Beyer, H.-G., Bongard, J., Branke, J., Clark, J.A., Cliff, D., Congdon, C.B., Deb, K., Doerr, B., Kovacs, T., Kumar, S., Miller, J.F., Moore, J., Neumann, F., Pelikan, M., Poli, R., Sastry, K., Stanley, K.O., Stutzle, T., Watson, R.A., Wegener, I. (eds.) GECCO 2007: Proceedings of the 9th annual conference on Genetic and evolutionary computation, 7-11 July, vol. 2, pp. 1588–1595. ACM Press, New York (2007)
5. Soule, T., Foster, J.A.: Effects of code growth and parsimony pressure on populations in genetic programming. Evolutionary Computation 6(4), 293–309 (1998)
6. Luke, S., Panait, L.: A comparison of bloat control methods for genetic programming. Evolutionary Computation 14(3), 309–344 (2006)
7. Langdon, W.B.: Size fair and homologous tree genetic programming crossovers. Genetic Programming and Evolvable Machines 1(1/2), 95–119 (2000)
8. Crawford-Marks, R., Spector, L.: Size control via size fair genetic operators in the PushGP genetic programming system. In: Langdon, W.B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M.A., Schultz, A.C., Miller, J.F., Burke, E., Jonoska, N. (eds.) GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, 9-13 July, pp. 733–739. Morgan Kaufmann Publishers, San Francisco (2002)
9. Poli, R.: A simple but theoretically-motivated method to control bloat in genetic programming. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E.P.K., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 204–217. Springer, Heidelberg (2003)
10. Zhang, B.-T., Mühlenbein, H.: Evolving optimal neural networks using genetic algorithms with Occam's razor. Complex Systems 7, 199–220 (1993)
11. Zhang, B.-T., Mühlenbein, H.: Balancing accuracy and parsimony in genetic programming. Evolutionary Computation 3(1), 17–38 (1995)
12. Zhang, B.-T., Ohm, P., Mühlenbein, H.: Evolutionary induction of sparse neural trees. Evolutionary Computation 5(2), 213–236 (1997)
13. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer, Heidelberg (2002)
14. Poli, R., McPhee, N.F.: 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: Miller, J., Tomassini, M., Lanzi, P.L., Ryan, C., Tetamanzi, A.G.B., Langdon, W.B. (eds.) EuroGP 2001. LNCS, vol. 2038, Springer, Heidelberg (2001)
15. Poli, R., Langdon, W.B., Dignum, S.: On the limiting distribution of program sizes in tree-based genetic programming. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, Springer, Heidelberg (2007)
16. Rosenfeld, A., Kak, A.C.: Digital Picture Processing, vol. 1,2. Academic Press, London (1982)
17. Koza, J.R. (ed.): Genetic Programming: On the Programming of Computers by Means of Natural Selection, USA, MIT Press, Cambridge (1992)
18. Iba, H.: Random tree generation for genetic programming. In: Ebeling, W., Rechenberg, I., Voigt, H.-M., Schwefel, H.-P. (eds.) PPSN 1996. LNCS, vol. 1141, pp. 144–153. Springer, Heidelberg (1996)

Practical Model of Genetic Programming's Performance on Rational Symbolic Regression Problems

Mario Graff and Riccardo Poli

Department of Computing and Electronic Systems, University of Essex, UK
{mgraff,rpoli}@eseex.ac.uk

Abstract. Many theoretical studies on GP are criticized for not being applicable to the real world. Here we present a practical model for the performance of a standard GP system in real problems. The model gives accurate predictions and has a variety of applications, including the assessment of the similarities and differences of different GP systems.

1 Introduction

Despite recent successes in developing solid theoretical foundations for Genetic Programming (GP) (e.g., see [3,6] and the recent review in [5]) and the establishment of a forum where theoreticians and practitioners can meet and discuss (the "Genetic Programming Theory and Practice" workshop series [8,4,9,7]), there is a growing gap between GP theory and practice. Often theoretical studies in GP (and evolutionary computation more generally) are criticized for being rarely applicable to realistic situations. One of the reasons for this is that producing a comprehensive theory for complex adaptive systems such as evolutionary algorithms is objectively a very hard and slow process, while GP technology develops at an unrelentingly fast pace. Still, sometimes theoreticians appear to focus on approaches and problems that are too distant from practice. On the other hand, despite the proven effectiveness of GP, there is a growing need for a theory that can clarify the applicability of different types of GP to particular problems, provide design guidelines and, thereby, avoid the current, very time-consuming, practice of hand-tuning algorithms, parameters and operators.

In this paper we start filling this theoretical gap by proposing a practical model of GP which, unlike previous research, does not attempt to capture all the characteristics of the algorithm nor to model it exactly. Instead, we focus on modelling the most important characteristic from the practitioner's point of view: GP's performance. In particular, inspired by recent research where we successfully modelled the performance of a GA exploring small landscapes using a linear function [1], here we attempt to model the performance of GP using a linear approach. The main difference, here, however, is that we don't model a toy situation with tiny landscapes and unrealistically small populations. Instead, we model *standard GP* on a *huge class* of realistic problems: symbolic regression problems whose target model is a rational function.

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 122–133, 2008.
© Springer-Verlag Berlin Heidelberg 2008

The paper is organised as follows. In Section 2 we briefly review the main concepts presented in [1]. In Section 3 we identify problems with the original approach of [1] when applied to GP, and propose ways of going beyond them. In Section 4 we describe the parameter settings and problems used to test our approach. This is followed by our experimental results (Section 5) and some conclusions (Section 6).

2 Information Landscapes and GA Performance

Let $\Omega = \{0,1\}^\ell$ be the search space explored by a GA, and f be a fitness function over Ω. Let us enumerate the bit strings in Ω using their value in decimal. The information landscape [1] for a problem is represented by a comparison matrix M with elements

$$m_{i,j} = \begin{cases} 1 & \text{if } f(i) > f(j) \\ 0.5 & \text{if } f(i) = f(j) \\ 0 & \text{if } f(i) < f(j). \end{cases} \quad (1)$$

Because of obvious symmetries we more concisely represent the information landscape using the vector $\mathbf{v} = (m_{0,1}, m_{0,2}, ..., m_{2^\ell-2,2^\ell-1})$, which contains the elements of M above the main diagonal. This representation of f is particularly suited for search algorithms which only use relative fitness values to make decisions. For example, a GA based on rank or tournament selection only cares about which solution is superior to which other, not by how much.

Irrespective of the particular type of performance measure chosen, the performance of a GA is a function $P : f \to \mathbb{R}$. [1] proposed to approximate $P(f)$ using a semi-linear model in which f is first transformed into its corresponding information landscape, and then the components of the landscape are used in a linear function to approximate $P(f)$. That is

$$P(f) \approx c_0 + \mathbf{c} \cdot \mathbf{v}(f), \quad (2)$$

where c_0 is a suitable scalar constant, $\mathbf{c}$ is a suitable constant vector (termed the performance landscape), $\mathbf{v}(f)$ is the information landscape associated to f and $\cdot$ is the scalar product. The coefficient c_0 and the vector $\mathbf{c}$ are estimated using multivariate linear regression, i.e., by least square fitting of a training set of $(\mathbf{v}(f), P(f))$ pairs. The number of parameters to be estimated is $2^{2\ell-1} - 2^{\ell-1} + 1$, which grows exponentially with ℓ. So, exponentially large training sets of fitness functions are required. In principle, this is not an obstacle since there are many more distinct fitness functions than parameters to identify. However, since the GA is stochastic, for each fitness function the exact value of $P(f)$ (required to form an example $(\mathbf{v}(f), P(f))$ pair for the training set) is unknown and can only be estimated by running the GA many times with fitness function f. This makes the approach very costly indeed, and, for this reason, it has only been tested on very small cases ($\ell = 3$). In some cases, however, the linear approximation in Equation (2) has been exceptionally accurate. For example, when computed for a tournament-based GA and tested against *all* possible problems with $\ell = 3$, the linear model gave a correlation coefficient of over 93%.

3 Modelling GP Performance

Given its simplicity and successful results we attempted to extend the approach described in [1] and summarised in the previous section to predict GP performance. However, a number of serious problems effectively prevented this. So, a generalisation of the original approach became necessary. In Section 3.1 we describe these problems. In Section 3.2 we describe the new approach we take in this paper to developing performance models of GP. Finally, in Section 3.3 we explain how our models are adapted to optimise their accuracy.

3.1 Problems with Information Landscapes in GP

The first obstacle to using Equation (2) is, of course, the fact that, in principle, GP explores an infinite search space. So, one would need to estimate an infinite-dimension $\mathbf{c}$ vector, which would require an infinite-dimension training set. To rectify this problem, we considered limiting the size of the structures considered. After all, one might argue, Equation (2) is an approximation. So, what if we limited the M matrix (and, correspondingly, the $\mathbf{v}$ and $\mathbf{c}$ vectors) to include only comparisons between the fitness of program trees of depth $\mathcal{D}$ or less?

Unfortunately, for typical primitive sets, the number of distinct programs of up to a certain depth grows doubly exponentially with the depth, making it difficult to use the approach described in the previous section even for the smallest of trees. For example, there are 302 distich programs of depths 0 to 2 for the primitive set $\{x, y, \sqrt{\ }, +, \times\}$ and 6,087 programs of depth 0 to 2 for the primitive set $\{x_1, x_2, x_3, AND, NAND, OR, NOR\}$. This requires 45,452 and 18,522,741 parameters to be identified in Equation (2), respectively. These numbers are also the minimum number of fitness-performance pairs required in the training sets to make the identification of c_0 and $\mathbf{c}$ well-posed. The number of runs required to compute such training sets would then be of the order of millions and billions, respectively, making the approach really hard to port to GP.

Furthermore, the scalability of the information landscape approach is not the only problem to be faced when attempting to used it for GP. Another problem is that in most primitive sets there are symmetries which imply that two syntactically different trees may in fact present the same functionality. For example, within the search space generated by $\{x, y, \sqrt{\ }, +, \times\}$, the programs `(`$\sqrt{\ }$`(`$\times$` x y))` and `(`$\sqrt{\ }$`(`$\times$` y x))` are functionally indistinguishable. Even worse is the situation with the primitive set $\{x_1, x_2, x_3, AND, NAND, OR, NOR\}$, where, because we only have 3 variables, there can just be 256 different Boolean functions of these variables. So, each program in the search space must implement one of these 256 functions. Now consider two particular programs that implement the same functionality. Then, for all functions f considered in the training set, these two programs will produce identical components in the corresponding $\mathbf{v}(f)$ vectors. This creates dependent rows in the system of equations that one needs to solve to identify c_0 and $\mathbf{c}$. Unless, there are sufficient independent data-points, the system becomes impossible to solve due to numerical instability.

Indeed, all our attempts resulted in ill-conditioned problems. We then considered overcoming this problem by representing only the elements of the search space that implement different functionalities in the comparison matrix M (and the vector $\mathbf{v}$). However, this too was ridden with difficulty. For example, for the Boolean primitive set mentioned above, we would have 256 different trees, but there are 257 variables to identify. As a result, this approach, too, consistently gave ill-conditioned systems.

What is necessary is to generalise and then abandon the original information landscape representation. We do this in the next section.

3.2 Beyond Information Landscapes

In a sense the information landscape is a re-representation of the original fitness function. Indeed, to any searcher that uses only relative fitness to decide where to search next, this is an exact re-representation, since the searcher cannot make use of any information about a fitness function f other than that explicitly stored in the information landscape $\mathbf{v}(f)$. However, $\mathbf{v}(f)$ is not the only model of f one could use.

Formally the fitness function f is fully specified if one indicates what fitness is associated to each program in the search space. Here we propose an approximate re-representation of the fitness function, namely one where f is represented using the ordered set $\mathcal{F}(f) = \{f_1, f_2, \dots\}$ where $f_i \in \mathbb{R}$ is the fitness of program $i \in S$ (when the fitness function is f), where S is a subset of Ω, the set of all program trees constructed by recursively composing the primitives in the primitive set.[1] E.g., S might include all programs of up to a certain depth or size or might simply be an random sample of Ω. Then we propose to approximate the performance function of a GP system using the following linear function

$$P(f) \approx a_0 + \sum_{i \in S} f_i a_i. \tag{3}$$

Typically, in GP the f_i are computed as follows

$$f_i = \sum_{x \in \gamma} d(i(x), g(x)) \tag{4}$$

where $i(x)$ is the value returned by program i given input x, $g(x)$ is the target value in x, γ is a set of inputs where functionalities are tested, i.e., $\{(x, g(x)) : x \in \gamma\}$ is the set of fitness cases,[2] and the function d can be any reasonable comparison function. Commonly, $d(i(x), g(x)) = (i(x) - g(x))^2$ or $d(i(x), g(x)) = |i(x) - g(x)|$. In this work we define

$$d(i(x), g(x)) = \frac{(i(x) - g(x))^2}{|\gamma|}, \tag{5}$$

where $|\gamma|$ represents the cardinality of the set γ.

[1] Clearly, if $S \equiv \Omega$ then $\mathcal{F}(f)$ is an exact representation of f.

[2] Note, here we are representing the target values using a function notation, $g(x)$. This is because in many test environments the data we want GP to fit are obtained by sampling some ideal function $g(x)$ for $x \in \gamma$. This is what we will do here as well. However, the approach does not require the explicit knowledge of $g(x)$ for $x \notin \gamma$.

As we did for the performance landscape in Section 2, the coefficients a_i can be obtained by the least square method applied to a suitable training set of $(\mathcal{F}(f), P(f))$ pairs. The procedure to estimate $P(f)$, as before, requires running GP on problem f multiple times and estimating GP's performance by averaging (or some other statistical means).

A relevant question then is: How accurate is our new linear approximation of GP's performance? Also, presumably some choices of S are better than others. How can we choose the best S? We will look at these issues in the next section.

3.3 Model Optimisation

Before we describe the methodology used to test the linear model and to select the elements of the set S, we need to introduce a quality measure for different performance models. The Relative Squared Error is used:

$$rse = \frac{\sum_i (p_i - \tilde{p}_i)^2}{\sum_i (p_i - \bar{p})^2} \tag{6}$$

where i ranges over a set of test problems used to evaluate the accuracy of a model, p_i is the average performance recorded for problem i when performing real runs, $\tilde{p}_i$ is the performance predicted by a linear model, and $\bar{p}$ is the average mean performance over all runs. The objective is to obtain rse as close as possible to 0.[3]

We are now in a position to try to optimally identify the elements of set S to be used in our model of GP performance (Equation (3)). In order to obtain the elements of set S we use a Genetic Algorithm (GA) where each individual includes n loci, and where each allele represents a rational function (created with same procedure used to create the functions in the training and validation sets). Each GA individual, therefore, encodes a potentially different S set of size n. The objective is to find the set of n trees which gives the minimum rse. In some sense, therefore, each GA individual is a linear model. So, to evaluate each individual, first we need to derive the corresponding a_i coefficients. These can be obtained by using the least square method on a suitable training set. Once the a_i's are known we can predict GP's performance using Equation (3). Naturally, when the GA optimisation is over, one needs to test the best model (S) evolved using an independent validation set to ensure the model generalises correctly. The quality of the model is given by Equation (6).

Note that, while we said that S is subset of Ω, a simple inspection of Equations (3)– (5) reveals that nothing would prevent the inclusion in S of functions which are not elements of Ω. Both ways of building S lead to valid re-representations of f. In this paper we will focus on the case $S \subset \Omega$, but we will compare the advantages and disadvantages of the two ways of building S in future research.

[3] A value of rse close to 1 means that the model is as good (or bad) at predicting performance differences as the mean. A value of rse less than 1 means that the model predicts better than the mean, while $rse > 1$ implies worse predictions than the mean.

Table 1. Parameters used in the GP experiments

Crossover rate p_{xo}	100%, 90%, and 80%
Mutation rate p_m	0%, 10%, and 20%
Population size	1000
Number of generations	50

4 Test Problems and Parameter Settings

To test our performance model, we used a standard GP system (Table 1 shows the parameters of the GP system). We used subtree crossover and subtree mutation. In subtree mutation we generated random trees with a the maximum depth of 4. The only significant difference w.r.t. the GP system used in [2] is that we select crossover and mutation points uniformly at random, while [2] used different probabilities for internal nodes and leaves.

The benchmark problem used was continuous symbolic regression. The target functionality was produced by sampling rational functions. We created 200 different rational functions using the following procedure. Two polynomials, $W(x)$ and $Q(x)$, were built by randomly choosing the degree of each in the range 2 to 8, and then choosing random real coefficients in the interval $[-10, 10]$. The rational function is then given by $g(x) = \frac{W(x)}{Q(x)}$. Each of the 200 rational functions was then sampled at 21 points uniformly distributed in the interval $[-1, 1]$. These 21 points formed the set γ.[4] The resulting set of 200 problems was divided into a training set and a validation set. For each g in either set we performed 50 independent GP runs recording the fitness of the best individual found in each run. We then associated to each g an estimate of $P(f)$ obtained by averaging the best fitness values f_{best} recorded in the 50 runs. Since f is a measure of distance between the target g and a program i, values of $P(f)$ close to zero represent good performance.

The training set was used for the optimisation of S via the GA. We test the model in different configurations, but in all cases the cardinality of set $|S|$, n, was set to 5 and all the members of S were rational functions. In order to understand the behaviour of the model in different GP systems the crossover rate was varied from 80% to 100% and the mutation rate from 0% to 20%.

In this paper we used the class of rational symbolic regression problem as a benchmark. However, the methodology presented can be applied to others problems. In the future, for example, we intend to test this approach on the class of Boolean problems. Also, the accuracy of fit is not the only performance measure of interest. In future work we will also look at modelling success rates.

To give an idea of the time needed to build the model, it is necessary to remember that one needs to build a training set and a validation set, and these are created by running GP several times on each problem. Once these sets are built, the runtime of the GA is negligible. Our Python GA implementation takes only a few seconds to produce a model.

[4] We sample the functions in our dataset since we are only interested in their value for $x \in \gamma$ (see Equations (3)– (5)).

Table 2. Accuracy of the model for different configurations of the GP system

GP System p_{xo}	p_m	Training set rse	Validation set rse
100 %	0 %	0.0833	0.1960
90 %	10 %	0.1235	0.2092
90 %	0 %	0.0931	0.1995
80 %	20 %	0.1303	0.2389
80 %	10 %	0.1195	0.2731
80 %	0 %	0.0933	0.2172

5 Results

We performed 10 independent runs of the GA to obtain S on training data collected with a GP system with $p_{xo} = 90\%$ and $p_m = 0\%$. The mean rse at the end of the GA runs was 0.13029 and the standard deviation was 0.04517, indicating that in all runs the GA was able to find models that fitted our problem training-set well. As our final linear performance model we used the S set that exhibited the best rse both in the training set and in the validation set. The performance exhibit in the validation set by this model is 0.1995.

Since the model was selected looking at its performance in the validation set, we need to create a separate test set to assess the generalisation capability of model. This new set was created using the same procedure as for the training and validation sets. The performance exhibited by the model in this new set was 0.2353.

Table 2 shows the model's rse in the different GP configurations. As can be seen from the table, in all the configurations tested the model exhibit good accuracy in the training and validation sets. This implies that the elements of S, which were selected using the configuration $p_{xo} = 90\%$ and $p_m = 0\%$, can be used to model the different GP configurations presented in this work without the need to find a different S for each GP configuration.

In order to give an idea of the model's accuracy, in Figure 1(a) we show a scatter plot of the performance measured in actual runs vs the performance obtained via Equation (3). The data plotted correspond to the training set used to choose S (i.e., $p_{xo} = 90\%$ and $p_m = 0\%$). The solid line in the plot represents the behaviour of a perfect model. Figure 1(b) presents the same data as Figure 1(a) but with a different scale, so as to provide a clearer idea of the accuracy of the model for the functions where GP is able to find solutions that fit rational functions reasonably well. As can be seen, in both plots the points form a cloud around the perfect model which is a clear qualitative indication that Equation (3) is an accurate and unbiased model of GP performance.

Another way to measure the model's accuracy is to compare the statistics of the distribution of GP performance (for the training and validation sets) obtained by running the GP system against those obtained by using the model. Table 3 presents statistics of the GP system (for $p_{xo} = 90\%$ and $p_m = 0\%$)

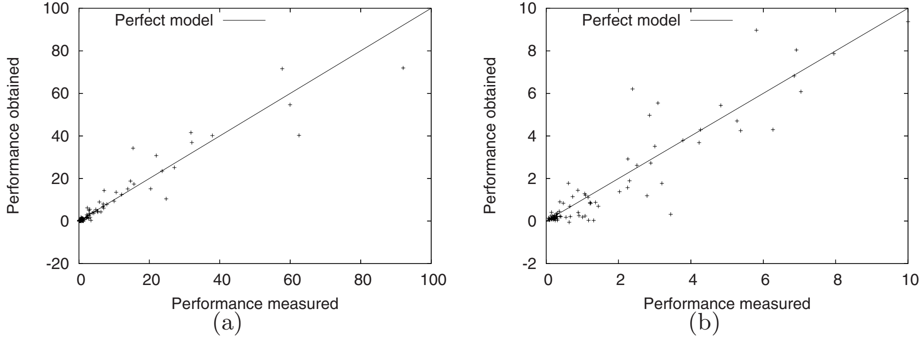


Fig. 1. Measured performance vs performance obtained using the linear model

in the training and validation set. The first and third columns are the statistics obtained by running the GP system and the second and fourth columns show the corresponding statistics obtained by using the model. While it is not surprising to find that the means in the training set are identical, we note that also the means in the validation set are very similar. Also the standard deviations are very similar, with noticeable discrepancies appearing only for higher order statistics and, particularly, for the validation set.

Together these results indicate that our linear model is really accurate. However, the differences in predicted vs actual kurtosis in the validation set caught our attention. This difference suggested that there might be a particularly unusual regression problem in the validation set where the GP system produced exceptionally bad performance. Indeed we found such a problem. Due to its rarity problems of this kind were not included in the training set and the model cannot predict performance accurately for this problem. Indeed, when we removed this problem from the validation set, we obtained that the new measured kurtosis in the validation set is 16.4639 and the one obtained by the model is 12.2223. So, all statistics up to the forth order match very closely.[5]

So far, in this section we have focused on the evaluation of the accuracy of our linear model of performance. In the rest of this section, we will describe the model obtained. As we will see there are many lessons one can learn from analysing it.

As is clear in Equation (3), a model is made up of two things: a set of functions S and the linear regression coefficients a_i associated to those functions. Table 4 shows the values of the a_i for different configurations of the GP system. The

[5] It is worth noting that while the model's accuracy in this rare problem was not very good, the model still predicted correctly that GP would have performed very badly on the problem. Indeed, the actual performance measure is 182.17 while the model predicted 104.70. Naturally, if accurately predicting the performance of rare problems where GP produces extremely bad results is important to a user, one could actively look for such problems and make sure that a suitable set of representatives is included in the training set.

Table 3. Comparison of actual vs predicted GP performance statistics (for $p_{xo} = 90\%$ and $p_m = 0\%$) for the training and validation sets

	Training set		Validation set	
	Measured	Obtained	Measured	Obtained
Mean	7.2354	7.2354	7.2268	7.2948
σ	14.9689	14.2551	21.3410	17.8337
Skewness	3.3902	2.8377	6.2037	3.5837
Kurtosis	12.7442	8.2259	44.8033	13.0803

Table 4. Values of a_i for different configurations of GP

GP system p_{xo}	p_m	a_0	a_1	a_2	a_3	a_4	a_5
100 %	0 %	155.3130	−0.3701	−0.1120	0.3674	−0.7178	0.8730
90 %	10 %	128.1946	−0.2073	−0.1292	0.0083	−0.5262	0.8356
90 %	0 %	151.3647	−0.2945	−0.1726	0.1758	−0.6398	0.9027
80 %	20 %	122.6533	−0.1365	−0.1317	−0.1367	−0.4826	0.8134
80 %	10 %	105.9296	−0.2429	−0.0317	0.0132	−0.4411	0.7935
80 %	0 %	139.4810	−0.2707	−0.1878	−0.0185	−0.5154	0.9261

table is complemented by Figure 2 which shows the plots of the functions in S that correspond to a_1 through to a_5.

The signs of the coefficients a_i ($i > 0$) can be interpreted as follows. If $a_i < 0$ then in order to have a good performance the target function must be far from the function of S corresponding to a_i. In other words, the value of the l.h.s. of Equation (4), where g is the target and i is the functionality associated with a_i, should be as high as possible. If $a_i > 0$, then g should be as close as possible to i for good performance.

Looking at the a_i coefficients in Table 4, we can see, for example, that $a_5 > 0$ in all configurations, meaning that if a target function is similar to the function in Figure 2(b), then good performance should be expected. Inspecting Figure 2(b) we can understand why: the function is nearly constant and close to 0 almost everywhere, and so, if the function g is similar to it, it would be extremely easy for GP to come up with a constant or a low order polynomial that fits it reasonably well.

Another interesting functionality is that associate to the coefficient a_3 (which is the one with a peak near $x = -0.8$ on the left of the plot in Figure 2(b)). As shown in Table 4, not all GP configurations present a_3's of the same sign. This suggests that there are classes of problems where a GP system with, for example, 100% crossover and 0% mutation would perform very differently from a GP system with 80% crossover and 20% mutation. Finally, all other coefficients are always negative, with the a_4 being the largest single contributor.

By analysing in detail the signs of the model's coefficients and the shape of the functions in S, it is possible to infer what problems are particularly easy or hard for any specific GP system. We will perform an in-depth analysis of this

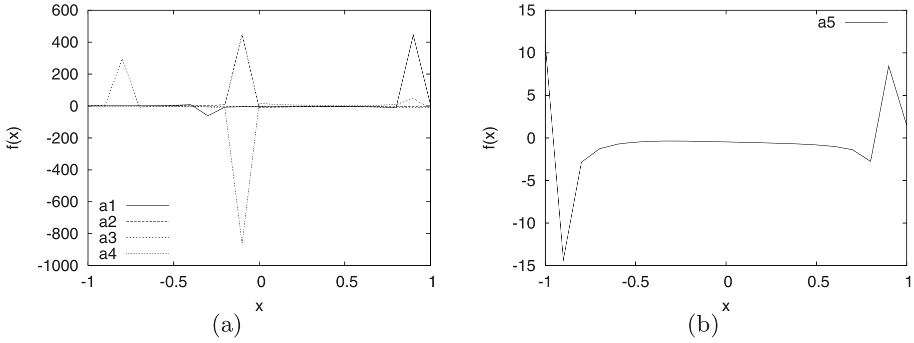


Fig. 2. Elements of set S

issue in future research. Here, instead, we will concentrate on another important applications of the model: the analytical comparison of different GP systems. As can be seen, Equation (3) represents an hyperplane. We propose to use the angle between the hyperplanes associated to different GP configurations to measure the difference in behaviour (in relation to performance) of such configurations. In order to obtain the angle between two hyperplanes we need to rewrite Equation (3) in its normal form, namely

$$(-1, a_1, \cdots, a_{|S|}) \cdot ((P, f_1, \cdots, f_{|S|}) - \bar{p}) = 0 \tag{7}$$

where $\bar{p}$ is a point on the hyperplane, and P is GP's performance. Using this representation the similarity between a GP system with the coefficients a'_i and a GP system with coefficients a''_i is the angle between vectors $(-1, a'_1, \cdots, a'_{|S|})$ and $(-1, a''_1, \cdots, a''_{|S|})$.

With this methodology we compared different configurations of GP, using configuration $p_{xo} = 100\%$ $p_m = 0\%$ as reference against which all other configurations are compared (via the calculation of the angle between each configuration and the reference). All configurations are shown as angles in Figure 3. The reference configuration is the x-axis. As shown in Table 5, the reference configuration has the worst mean performance. Interestingly, the configuration closest to the reference, $p_{xo} = 90\%$ $p_m = 0\%$, has the second worst mean, while the configuration that forms the largest angle with the reference, $p_{xo} = 80\%$ $p_m = 20\%$, presents the best mean.

Finally, we should note that the objective of this work is not to solve the set of rational symbolic regression problems well. That is we have used rational symbolic regression as a test bed to show the viability of linear models of GP performance. Nonetheless, looking at Table 5 any experience GPer will find it interesting that such problems are better solved by configurations of GP where mutation is used. There is also one other characteristic that is interesting: the standard deviation σ appears to decrease as the mutation rate increases. Are these differences statistically significant? If so, why do they occur? In future research we hope to be able to answer these questions.

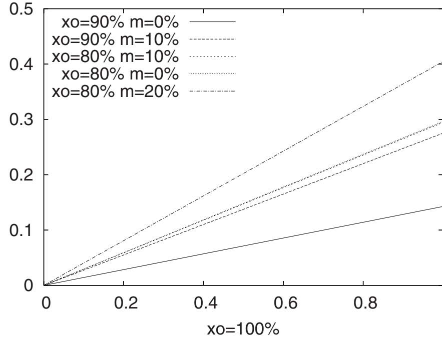


Fig. 3. Angle between GP with $xo = 100\%$ and the others settings

Table 5. Performance of the different GP systems

GP system	Training set		Validation set	
	Mean	σ	Mean	σ
$p_{xo} = 100\%$ $p_m = 0\%$	7.3591	15.6271	7.1952	20.7939
$p_{xo} = 90\%$ $p_m = 10\%$	6.5138	13.7010	5.6921	14.1050
$p_{xo} = 90\%$ $p_m = 0\%$	7.2354	14.9689	7.2268	21.3410
$p_{xo} = 80\%$ $p_m = 20\%$	5.9480	12.4507	5.4467	14.3367
$p_{xo} = 80\%$ $p_m = 10\%$	6.5210	13.8889	6.2925	18.4059
$p_{xo} = 80\%$ $p_m = 0\%$	7.1018	14.2824	7.0071	19.7771

6 Conclusions

In this paper we have presented a new practical model of the performance of GP. The model is unusual for two reasons: it models real (non-toy) GP systems in real (non-toy) problems, and it is extremely simple and accurate. We tested this approach on the class of rational symbolic regression problems.

The simplest and most obvious application of our model is to test whether a function is hard or easy for GP without actually running the system. Another application studied in this work is the comparison of different GP systems. We have made a first step in this direction. In future work we plan to do more extensive comparisons of GP systems with different parameters (e.g., crossover rates). In addition we want to compare the behaviour of tree-based, linear and grammar-based GP, just to mention a few. Also, in this work we analysed only the case where performance was the mean fitness of the best individuals of each run. In future work we plan to test this methodology using different measures of performance like, for example, the success rate.

Besides the application describe in this paper, there are many more which we did not have space to describe, and which will be the subject of future papers. For example, it is possible to use the model to directly create functions that are particularly hard or easy for GP. Another important application of the model

is to select which set of GP parameters, operators or algorithms would give the best results for a given symbolic regression problem.

References

1. Borenstein, Y., Poli, R.: Information landscapes. In: Beyer, H.-G., O'Reilly, U.-M. (eds.) GECCO, pp. 1515–1522. ACM, New York (2005)
2. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
3. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer, Heidelberg (2002)
4. O'Reilly, U.-M., Yu, T., Riolo, R.L., Worzel, B. (eds.): Genetic Programming Theory and Practice II. Genetic Programming, Ann Arbor, MI, USA, 13-15 May, vol. 8. Springer, Heidelberg (2004)
5. Poli, R., Langdon, W.B., McPhee, N.F., Koza, J.R.: Genetic programming an introductory tutorial and a survey of techniques and applications. Technical Report CES-475, Department of Computing and Electronic Systems (October 2007)
6. Poli, R., McPhee, N.F.: General schema theory for genetic programming with subtree-swapping crossover: Part II. Evolutionary Computation 11(2), 169–206 (2003)
7. Riolo, R.L., Soule, T., Worzel, B. (eds.): Genetic Programming Theory and Practice IV, Ann Arbor, May 11-13, 2007. Genetic and Evolutionary Computation, vol. 5. Springer, Heidelberg (2007)
8. Riolo, R.L., Worzel, B.: Genetic Programming Theory and Practice. In: Koza, J. (ed.) Genetic Programming, vol. 6, Kluwer, Boston, MA, USA (2003)
9. Yu, T., Riolo, R.L., Worzel, B. (eds.): Genetic Programming Theory and Practice III, Ann Arbor, May 12-14, 2005. Genetic Programming, vol. 9. Springer, Heidelberg (2005)

Semantic Building Blocks in Genetic Programming

Nicholas Freitag McPhee[1,2], Brian Ohs[1], and Tyler Hutchison[1,3]

[1] Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota
USA 56267
{mcphee,ohsx0004,hutc0125}@morris.umn.edu
[2] http://www.morris.umn.edu/~mcphee/
[3] http://www.tylersaurus.com

Abstract. We present a new mechanism for studying the impact of subtree crossover in terms of semantic building blocks. This approach allows us to completely and compactly describe the semantic action of crossover, and provide insight into what does (or doesn't) make crossover effective. Our results make it clear that a very high proportion of crossover events (typically over 75% in our experiments) are guaranteed to perform no immediately useful search in the semantic space. Our findings also indicate a strong correlation between lack of progress and high proportions of fixed contexts. These results then suggest several new, theoretically grounded, research areas.

1 Introduction

Subtree crossover is one of the oldest and remains one of the most widely used recombination operators in genetic programming (GP). It is still unclear, however, why or how it works. It's hardly obvious that yanking a random chunk of code from one program, and plopping it unceremoniously in a random location in a second program would be a good thing. Yet it clearly works (at some level) in GP.

But why? How does subtree crossover move the population closer to the solution? Is it really just a happy accident that this simple operator provides some sort of useful recombination? Are there better operators and representations waiting to replace this strangely random process?

Here we present a new mechanism for studying the *semantic* effect of subtree crossover in terms of *semantic building blocks*. Subtree crossover combines two tree components: the context (the root parent with a subtree removed) and the subtree being inserted into that context. Our approach allows us to completely and compactly describe (for Boolean problems) the semantics of these two key components (contexts and subtrees), which allows us to completely describe the semantic action of subtree crossover. We can also enumerate the occurrences of different context and subtree semantics in a population, independent of their syntax. This allows us to perform detailed studies of the semantic components present in a population, and the opportunities this provides for subtree crossover. The resulting data strongly suggest that the distribution of context semantics are key to the success (or failure) of runs. Our results also make it clear that

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 134–145, 2008.
© Springer-Verlag Berlin Heidelberg 2008

a very high proportion (typically over 75% in our experiments) of crossover events are *guaranteed* to perform no immediately useful search in the semantic space. These tools and results not only shed new light on the operation and impact of subtree crossover, but they also suggest a number of ideas for new operations and approaches to genetic programming based on this new theoretical and empirical understanding.

In the next section (Section 2) we review some of previous work on GP building blocks and the behaviour of crossover. In Section 3 we present our new tools and show how the semantics of contexts and subtrees can be calculated and enumerated. In Section 4 we go over results from empirical runs and data collected using new measures enabled by these ideas. We discuss those results and some of their implications in Section 5, and conclude in Section 6.

2 Related Research

"Building blocks" have a long history in genetic algorithms (GAs), and there have been various definitions proposed for building blocks in GP. These were typically strictly syntactic in nature, and often part of an effort to adapt GA schema theory to GP (e.g., [13,19,15,20,16,17]; see [4] for additional review). There have also been numerous studies on the impact of subtree crossover, other recombination operators, and their interactions with things like mutation [2,1,14,8,12,4].

Many of these studies have helped us better understand important properties of GP such as code growth. None, however, have shed much light on the underlying semantic behavior of subtree crossover or provided tools to track and analyze those semantics. Perhaps the closest to the current research that we're aware of is [2], where the proposed marking process captures useful semantic information about potential crossover points that is related to our notion of fixed contexts (discussed in Section 3.2).

3 Enumerating Semantic "Building Blocks"

In sub-tree crossover we construct a new offspring by replacing a randomly chosen sub-tree from parent A with a random sub-tree from parent B. To understand the possibilities afforded by sub-tree crossover, then, we need to be able to characterize what sub-trees can be chosen from B, and where they can go in A.

We define a *context* to be a tree with some specific (but arbitrary) subtree removed (see Figure 1); we will use '#' to indicate the removed subtree.[1] Given this definition, describing the semantic impact of sub-tree crossover reduces to describing the semantics of sub-trees, the semantics of contexts, and their interactions. We will describe these ideas in some detail below; see [10] for additional detail and examples.

This paper will focus on the Boolean domain, i.e., trees that represent Boolean functions. Working in such a small (finite) domain is valuable because it makes it much easier to compute and catalogue the complete semantics of the sub-trees and contexts involved. (See Sec 5 for more on extending these ideas to other domains.) In the Boolean

[1] This is similar to a tree schema with one '#' leaf symbol from [4]. A schema, however, represents a *set* of trees, whereas for us a context is simply a syntactic construct.

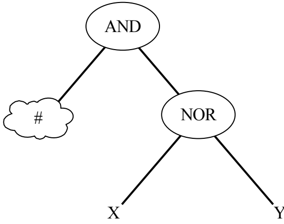


Fig. 1. An example of a context, i.e., a tree with one subtree (represented by `#`) removed

Table 1. The (sub)tree semantics for the four Boolean functions used in our experiments. In a finite (e.g., Boolean) domain we can fully characterize the semantics of a (sub)tree by enumerating the values of a tree (i.e., a function) on all its possible inputs.

x	y	(and x y)	(or x y)	(nand x y)	(nor x y)
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	0

domain we can generate a highly compact representation of both subtree and context semantics. This allows us to enumerate the semantics of all the sub-trees in a given tree in the population, or even all the sub-trees of *all* the trees in a given population. We can then explore this distribution of sub-tree semantics to better understand the possibilities available to sub-tree crossover.

3.1 Semantics of Subtrees

Following the ideas used in sub-machine code GP [18], we can completely specify the semantics of a Boolean valued (sub)tree (or, equivalently, function) by enumerating its value on each of the possible sets of input values (as in the construction of truth tables). Taking 0 to be *false* and 1 to be *true*, the function `(and x y)`, for example, has the semantics `0001` corresponding to the third column in Table 1. (See Fig. 2 and [10] for additional examples of sub-tree semantics.) This, then, allows for a complete characterization of the semantics of any Boolean (sub)tree in the sense that if two trees S_0 and S_1 have the same semantics, and tree T contains S_0 as a sub-tree, we can replace the occurrance of S_0 in T with S_1, and the semantics of T will remain unchanged.

3.2 Semantics of Contexts

In general we won't know the semantics of a tree with an unspecified subtree removed, since the details of that subtree will usually affect the semantics of the entire tree. However, some contexts depend less on the details of their open subtree than others. For example, the context `(and false #)` is *always* going to return *false*, regardless of which subtree we insert into the open position. Further, we know from experience that genetic programming has strong tendencies towards the creation of such contexts [9,5,4].

We refer to a context as being *fixed* for a particular set of inputs (or a particular *position* when using strings to represent semantics) if the value of that context is completely determined (either `true` or `false`) regardless of what subtree is inserted at the open node (`#`). We define the entire context to be fixed if it is fixed for *every* possible set of inputs (i.e., at every position in the semantics string).

In the Boolean domain the semantics of a context depend on the details of the inserted subtree in a systematic manner. Consider the context `(and true #)`. Here the value

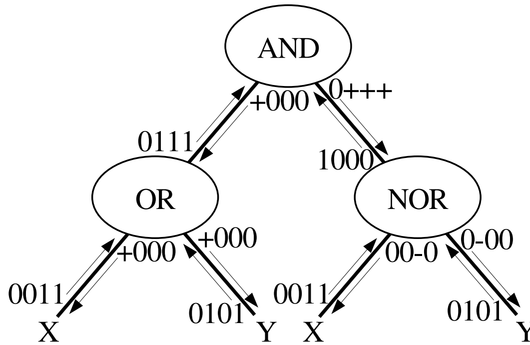


Fig. 2. A sample syntax tree showing both subtree and context semantics. The arrows pointing upward (on the left of the edges) are the semantics of the subtree below them, e.g., the semantics of `(or x y)` is `0111`. The arrows pointing downward (on the right on the edges) are the semantics of the context obtained by removing the subtree below the arrow. For example, the semantics of `(and # (nor x y))` is `+000`.

of this context will be the *same* as the value of whatever subtree we insert for the `#`. We will denote the semantics in such a case with a `+`, indicating that the value of the subtree passes through unchanged. The alternative case is represented by a context like `(nand 1 #)`. Here the value of the context is going to be the *negation* of the value returned by the inserted subtree. We will use a `-` to denote the semantics in this case. ([10] provides several examples in more detail.) Thus while the interactions between contexts and subtrees can be quite complex, in the case of Boolean functions there are only four options for a context on a specific set of inputs: the fixed semantics (`0` and `1`), the "unchanged semantics" (`+`), and the "negation semantics" (`-`).

A key difference between subtree semantics and context semantics is which components need to be taken into consideration when computing the semantics. The semantics of a subtree are solely a function of the operator and the value of its arguments; they are completely independent of where that subtree might be located. For context semantics, the case is slightly more complex. While we associate the semantics with the edge above the insertion point, they are still a function of the entire tree around that point. In particular they depend on three things (see Fig. 3):

- The operator g immediately above the insertion point.
- The semantics of the context obtained by removing the subtree rooted at g (the "Parent semantics" in Figure 3).
- The subtree semantics of the other argument (x) of the operator g (the "Arg semantics" in Figure 3).

The one exception is when the insertion point (`#`) is in fact the root of the context, in which case there is no parent node. In this case the context semantics are simply defined to be `+` since the value returned by the tree is going to be the value of the inserted subtree.

Table 2 lists the cases for the Boolean functions used in this work: *and*, *or*, *nand*, and *nor*. In the last line of Table 2, for example, if the parent semantics of a *nand* node is `-`, and the argument semantics of the sibling is `1`, then the context semantics is `+`.

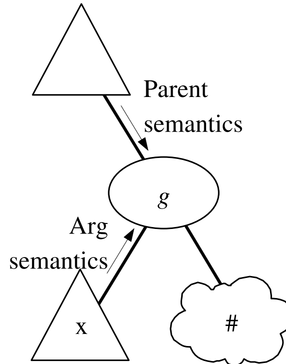


Fig. 3. Illustration of the interaction of the different components in computing the context semantics. Here we have a tree with some subtree removed (the insertion point, indicated by the # in the lower right). g is the parent node of the insertion point, and x represents the other argument of g (i.e., the sibling subtree of the insertion point). Note that x is not necessarily a leaf but can represent an arbitrarily complex node. The semantics of this context is then a function of the specific operator g, the semantics of the context obtained by removing the subtree rooted at g, and the subtree semantics of x.

Notationally it is convenient to associate context semantics with the edge extending *down* to the insertion point (i.e., the # symbol), as this allows us to indicate the semantics of all the possible contexts in a tree on a single diagram as is done in Figure 2. It's important to realize, however, that even though they are attached to a specific edge, these semantics describe the *entire* context, i.e., the entire tree minus the subtree below the edge in question.

4 Empirical Results

To see how subtree crossover affects the distribution of both context and subtree semantics, we did multiple runs on five different problems: Even Parity problems with 2, 3, 4, and 6 bits (2-EP, 3-EP, 4-EP, and 6-EP), the 6-bit multiplexer (6-MUX) problem, and flat fitness on four bits (4-Flat).

4.1 Parameters and Data Collected

For each problem we did 38 independent runs using the parameters listed in Table 3. Since we weren't particularly interested in maximizing our chances of solving the problems, no effort was made to tune our parameter choices.

For each test problem except the flat fitness case (4-Flat) the fitness was the number of test cases handled correctly, with higher values being better. For 4-Flat the fitness was constant for all individuals, so there was no selection bias in those runs.

Along with traditional data such as fitnesses and tree sizes, we also tracked two kinds of data specific to building block semantics:

Table 2. The context semantics for *and*, *or*, *nand*, and *nor*. These functions are all symmetric, so we only show the context with the # as the second argument. See the text for further details.

Parent semantics	Arg semantics (x)	`(and x #)`	`(or x #)`	`(nand x #)`	`(nor x #)`
0	0	0	0	0	0
0	1	0	0	0	0
1	0	1	1	1	1
1	1	1	1	1	1
+	0	0	+	1	-
+	1	+	1	-	0
-	0	1	-	0	+
-	1	-	0	+	1

Table 3. Parameters used in our runs. The crossover probability of 1 means that subtree crossover was the *only* recombination operator used in these runs, i.e., there was no mutation and no reproduction.

Parameter	**Value**
Function set	Binary AND, OR, NAND, and NOR
Terminal set	$x_0, x_1, \ldots, x_{n-1}$, where n is the number of variables (or bits) in the problem.
Control strategy	Generational
Population size	1000
Initialization	PTC2 [7], with equal proportions of sizes 50, 70, and 100 nodes and maximum initial depth of 10
# of generations	500
Tournament size	2
XO Probability	1
XO bias away from leaves	None (all nodes are equally likely)
Maximum size after XO	500 (If the resulting child is too large, then new parents are chosen independently and process begins again.)

Proportion of fixed contexts. The percentage of contexts (over all contexts in every individual in the population) that are completely fixed, i.e., all the positions are either a 0 or 1. This means that any crossover using this context is going to be a semantic no-op, yielding an offspring with the same semantics as the context regardless of the subtree inserted. High proportions of fixed contexts suggest that a run has essentially stalled, with very little effective search going on anymore. Note that this is not *necessarily* a bad thing – if the run has found the target, for example, then fixing strongly is not necessarily problematic. However, if the target has yet to be found then a large proportion of fixed contexts is probably undesirable.

Proportion of compatible contexts. The percentage of contexts (over all contexts in every individual in the population) that are *compatible* with the target context. A compatible context is a context that has the possibility of producing a target solution in one step, meaning that any fixed values in the context must match the

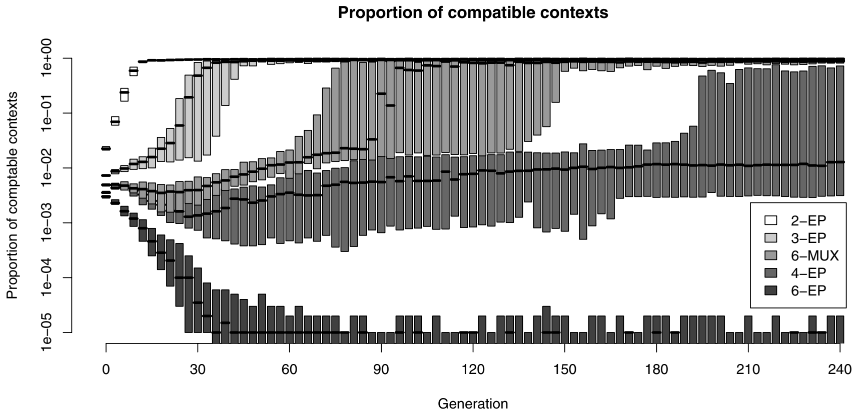


Fig. 4. Boxplots of the percentage of compatible contexts for the first 240 generations across the 38 runs for all five problems. The median and two middle quartiles are plotted. Note the log scale on the y-axis.

corresponding values in the target. (A fully fixed context that will always produce a target solution is also considered a compatible context.) If a context is incompatible, it is guaranteed to not produce a solution if it is used in a crossover event. Therefore low proportions of compatible contexts suggest that a run is unlikely to succeed, at least in the near term.

4.2 Results

Not surprisingly, the 2-EP and 3-EP problems were quite easy and had 100% success rates. The 4-EP runs found a solution 20 out of 38 times; two of these 20 runs, however, later lost their successful solution (we weren't using any form of elitism) and ultimately converged on functions with fitness 15. None of the 6-EP runs solved the problem, while all of the 6-MUX were successful, supporting the idea that 6-MUX is generally much easier to solve than 6-EP with this function set. We weren't particularly interested in the success of the runs, but the relative difficulty of these problems (as demonstrated by these success rates) is clearly reflected in many of the results below.

Compatible Contexts. Figure 4 plots the proportion of compatible contexts for all five non-flat problems (Flat fitness is not included here because it does not make sense to talk about compatible contexts when there is no target to be compatible with.) The proportion of compatible contexts for the relatively easy 2-EP and 3-EP problems quickly jumps to nearly 1 as solutions are found, and subsequent bloat leads to large trees with many (correctly) fixed contexts. 6-MUX, which also has a very high success rate, shows a similar behavior, although it takes a little longer for it to find a solution so the proportion of compatible contexts does not rise as soon.

The proportion of compatible contexts for most 6-EP runs quickly drops to effectively zero, indicating that those runs have converged on local optima that are

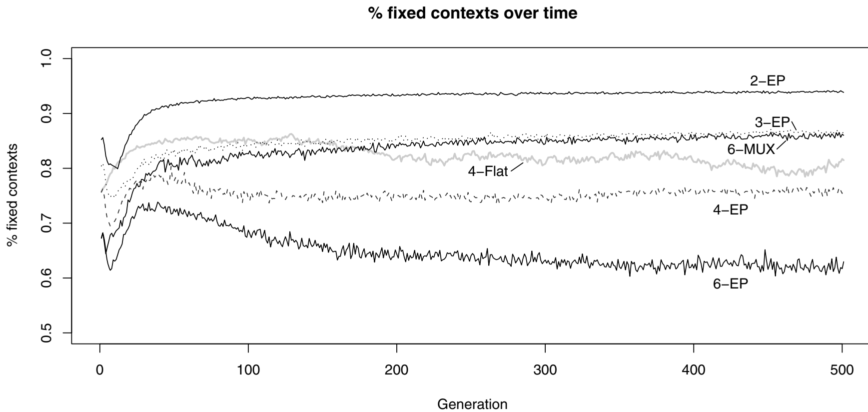


Fig. 5. Median proportion of contexts that are fixed vs. generation for all six test environments. Note the y-axis doesn't continue all the way down to 0. Also, the 3-EP and 6-MUX plots almost completely overlap for the second half of the plots.

inconsistent in a significant way with the target. This suggests that those runs are very unlikely to ever find a solution, as it would presumably take a significant jump to move from the peak they've converged on to the target peak. What's not indicated in the plot (because the whiskers and outliers are suppressed) is that there are handful of runs (4 of 38) with considerably higher proportions of compatible contexts. The proportions in these runs are around 0.01, putting them in the lower range of the plotted data for the 4-EP runs. None of the 6-EP runs succeed in finding a solution in the 500 generations we used, but it seems plausible that this small group of runs would be the most likely to eventually find a solution if given more time.

The higher persistent variance in the percentages for the 4-EP runs presumably reflects the fact that several of the runs have succeeded, while others remain stuck at local optima. The fact that many of the (so far) unsuccessful 4-EP runs have percentages of compatible contexts that are well above zero (around 0.01) suggests, however, that those runs may still have some chance of eventually jumping to the solution.

The percentage of compatible contexts provides an upper bound on the probability of constructing a target solution via sub-tree crossover (since doing so will require both a compatible context and an appropriate subtree). Our empirical data shows, at least for these test conditions, that the probability of constructing a target is in fact almost equal to the percentage of compatible contexts. (See [11] for more.)

Proportion of Fixed Contexts. Figure 5 shows the median proportion (over the 38 runs) of contexts that are *completely* fixed for each of our six test environments. Remember that a completely fixed context is one where the return value *in all cases* is completely determined and will be unaffected by the details of the particular subtree inserted into the context. Thus a crossover using a completely fixed context is guaranteed to generate an offspring with the same semantics as the root (or context) parent, and no semantic exploration will have occurred.

It is worrying, then, that in each of the six cases, the proportion of fixed contexts in the population exceeded 60% at all times, and was typically greater than 75%. This means that a great majority of all crossover events are (at least in the short term) useless, as they don't explore any new semantic space.[2] The harder problems (4-EP and 6-EP) had the lowest percentage of fixed contexts, but even in these problems well over half of all crossovers were guaranteed to be explore no new semantic space.

All the problems except 4-Flat show a small drop in the proportion of fixed contexts in the first few generations, with the proportion of fixed contexts then climbing again from around generation 10 to around generation 30; we suspect the dip is connected to the early reshuffling often seen in GP runs. After 200 generations the median proportion of fixed contexts become fairly flat in all problems, except 4-Flat where there appears to be a certain amount of drift. As was true with the proportion of compatible contexts, the percentage of fixed contexts also "sorts" the problems according to difficulty. The percentages of fixed contexts at generation 500 are in fact pairwise different at the 5% confidence level (using the pairwise Wilcoxon test) except for 4-Flat vs. 4-EP, and 3-EP vs. 6-MUX.

Figure 5 also shows that the proportion of fixed contexts in the initial generations are driven (for these problems) by the number of inputs. 4-EP and 4-Flat for example, both start at the same proportion, but diverge almost immediately, with 4-Flat's proportion of fixed contexts growing gently instead of dipping initially as is the case with 4-EP. Similarly 6-EP and 6-MUX start out similarly for a few generations, and then diverge as the 6-MUX runs begin to gain traction on the problem while the 6-EP runs continue to flounder.

5 Discussion

Approximation and Extension to Non-Boolean Domains. As mentioned in Section 3, these techniques can currently only be used on Boolean problems, and even for Boolean problems they scale badly with the number of variables. Of the new measures mentioned above (proportions of compatible contexts and proportion of fixed contexts), the proportion of fixed contexts is probably the easiest to generalize to problems with more variables and non-Boolean domains. Given that the proportion of fixed contexts appears to have potential as an indicator of problem difficulty (see Figure 5), being able to estimate it should have value.

One could estimate the proportion of fixed contexts on larger Boolean problems, for example, by randomly sampling contexts (perhaps as part of the existing crossover process), and checking to see if they're fixed. One could, for example, insert each of the 2^{2^N} different subtree semantics at the crossover point in the context to see if any changed the semantic value of the context. It is sufficient, however, to only check any two complementary subtree semantics (e.g., the constants *true* and *false*). If the (Boolean) context is in fact not completely fixed, then it must contain at least one '+' or '-', which means it will have different values for at least one set of inputs when complementary subtree

[2] It is possible that such a crossover creates new syntactic structure that, when sampled in later generations, will lead to an important discovery. Given the lack of any immediate semantic effect, however, such benefit is quite random and unguided by the fitness function.

semantics are inserted. One would still need to check all 2^N possible inputs to know for certain if the context is fixed, but for large N one could further approximate by sampling the set of possible inputs.

For non-Boolean domains the problem becomes more complex, especially in continuous domains like symbolic regression over the reals. With real-valued functions, for example, there is potentially a whole spectrum of fixation. A completely fixed context might (as in the Boolean case) be completely independent of the inserted subtree, while a "nearly fixed" context might change, but only by very small amounts. There's also no simple analogy to the complementary subtree semantics (such as *true* and *false*) to simplify the sampling of the subtree semantics. Still, it seems likely that sampling a few constant values at the insertion point across several sets of input values would provide a useful approximation of the "fixedness" of a context, even in a real-valued problem.

Designing New Operators and Representations. In many ways the high proportion of fixed contexts (Figure 5) is quite disheartening, as it suggests that the majority of crossover events are exploring no new semantic space. We could, therefore, use these results to guide the design of new recombination operators that would deliberately work to reduce the proportion of fixed contexts, hopefully increasing the exploratory power of our system.

Experiments, for example, with a crossover operator that avoids choosing approximately fixed contexts (essentially the same as the approach taken in [2]) don't appear to improve the likelihood of finding solutions and can significantly slow down the evaluation of individuals. It does, however, provide a very effective bloat control mechanism, and it's possible that modifications of this idea, or combinations with other operators, could improve performance.

We could see the data reported here as the result of a co-evolutionary system where there is a serious problem of *disengagement* [3], where one population (the contexts) "beats" the other (the subtrees). In this case the population of contexts reaches such a high proportion of fixedness that the subtrees are essentially frozen out of the process. The restricted crossover operator defined above, then, could be seen as a means of combating disengagement by increasing the chances that a context distinguishes among subtrees instead of simply dominating them [3]. One could extend this observation to build an explicit co-evolutionary model of subtree crossover in GP. Obviously in standard GP the "population" of contexts and the "population" of subtrees are linked on several levels (any particular node is a component of numerous contexts and numerous subtrees at the same time), but given the apparent dominance of contexts in determining the likelihood of success, detaching the two might in fact prove helpful rather than problematic.

Alternatively, one could see our distributions of context and subtree semantics as the basis for a co-evolutionary estimation of distribution (EDA) algorithm [6].

6 Conclusions

In this paper we have presented a novel means of exactly and compactly describing (for Boolean problems) the semantics of the two tree components combined by subtree

crossover: the context (the root parent with a subtree removed) and the subtree being inserted into that context. This allows us to completely describe the semantic action of subtree crossover, and enumerate in a syntax independent fashion the occurrence of different context and subtree semantics in a population. The resulting data strongly suggest that the distribution of context semantics are key in the success (or failure) of runs. The proportion of fixed contexts in these problems is very high (typically over 75%), indicating that the substantial majority of subtree crossover events actually perform no search in the semantic space.

As well as shedding valuable new light on the impact of subtree crossover, these tools and results suggest a number of ideas for new operations and approaches to genetic programming that would be based on theoretical and empirical understanding rather than simple guesswork.

Acknowledgments

Nic would like to thank the organizers and participants in Dagstuhl Seminar 06061 for providing a venue to present early versions of this work, for valuable feedback and ideas. Nic would also like to thank Riccardo Poli and the University of Essex for providing such a productive sabbatical environment. Brian and Tyler gratefully acknowledge funding from both the Morris Academic Partners Program and the University of Minnesota Undergraduate Research Opportunities Program.

References

1. Angeline, P.J.: Subtree crossover: Building block engine or macromutation? 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, July 13-16, pp. 9–17. Morgan Kaufmann, San Francisco (1997)
2. 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, pp. 33–38 (1994), Max-Planck-Institut für Informatik (MPI-I-94-241)
3. de Jong, E.D., Pollack, J.B.: Ideal evaluation from coevolution. Evolutionary Computation 12(2), 159–192 (2004)
4. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer, Heidelberg (2002)
5. Langdon, W.B., Soule, T., Poli, R., Foster, J.A.: The evolution of size and shape. In: Spector, L., Langdon, W.B., O'Reilly, U.-M., Angeline, P.J. (eds.) Advances in Genetic Programming 3, ch. 8, pp. 163–190. MIT Press, Cambridge (1999)
6. Larrañaga, P., Lozano, J.A.: Estimation of Distribution Algorithms, A New Tool for Evolutionary Computation. Kluwer Academic Publishers, Dordrecht (2002)
7. Luke, S.: Two fast tree-creation algorithms for genetic programming. IEEE Transactions on Evolutionary Computation 4(3), 274–283 (2000)
8. Luke, S., Spector, L.: A revised comparison of crossover and mutation in genetic programming. 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, July 22-25, pp. 208–213. Morgan Kaufmann, San Francisco (1998)

9. McPhee, N.F., Miller, J.D.: Accurate replication in genetic programming. In: Eshelman, L. (ed.) Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA 1995), Pittsburgh, PA, USA, July 15-19, pp. 303–309. Morgan Kaufmann, San Francisco (1995)
10. McPhee, N.F., Ohs, B., Hutchison, T.: Enumerating building block semantics in genetic programming. Technical report, University of Minnesota, Morris (2007), http://www.morris.umn.edu/academic/fclt/WorkingPapers/Morris_WP_3.1.pdf
11. McPhee, N.F., Ohs, B., Hutchison, T.: Semantic building blocks in genetic programming. Technical report, University of Minnesota, Morris (2007), http://www.morris.umn.edu/academic/fclt/WorkingPapers/Morris_WP_3.2.pdf
12. McPhee, N.F., Poli, R.: Using schema theory to explore interactions of multiple operators. In: Langdon, W.B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M.A., Schultz, A.C., Miller, J.F., Burke, E., Jonoska, N. (eds.) GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, July 9-13, pp. 853–860. Morgan Kaufmann Publishers, San Francisco (2002)
13. O'Reilly, U.M., 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)
14. Poli, R.: Is crossover a local search operator? In: Position paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-1997, July 20 (1997)
15. Poli, R., Langdon, W.B.: Schema theory for genetic programming with one-point crossover and point mutation. Evolutionary Computation 6(3), 231–252 (1998)
16. Poli, R., McPhee, N.F.: General schema theory for genetic programming with subtree-swapping crossover: Part I. Evolutionary Computation 11(1), 53–66 (2003)
17. Poli, R., McPhee, N.F.: General schema theory for genetic programming with subtree-swapping crossover: Part II. Evolutionary Computation 11(2), 169–206 (2003)
18. Poli, R., Page, J.: Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code GP and demes. Genetic Programming and Evolvable Machines 1(1/2), 37–56 (2000)
19. Rosca, J.P.: Analysis of complexity drift 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, July 13-16, pp. 286–294. Morgan Kaufmann, San Francisco (1997)
20. Sastry, K., O'Reilly, U.-M., Goldberg, D.E., Hill, D.: Building block supply in genetic programming. In: Riolo, R.L., Worzel, B. (eds.) Genetic Programming Theory and Practice, ch. 9, pp. 137–154. Kluwer, Dordrecht (2003)

A Simple Powerful Constraint for Genetic Programming

Gearoid Murphy and Conor Ryan

Biocomputing and Developmental Systems
Computer Science and Information Systems
University of Limerick, Ireland
{gearoid.murphy,conor.ryan}@ul.ie
http://bds.ul.ie

Abstract. This paper demonstrates the ability of Hereditary Repulsion to perform well on a range of diverse problem domains. Furthermore, we show that HR is practically invulnerable to the effects to overfitting and does not suffer a loss of generalisation, even in the late stages of evolution. We trace the source of this high quality performance to a pleasingly simple constraint at the heart of the HR algorithm. We confirm its effectiveness by incorporating the constraint into one of the benchmark systems, observing substantial improvements in the quality of generalisation in the evolved population.

1 Introduction

This paper demonstrates the striking effectiveness of an elegantly simple constraint at the heart of the Hereditary Repulsion algorithm. Hereditary Repulsion [10] is a convergence manipulation algorithm, which has been shown to dramatically improve the performance of GP on a difficult regression problem by improving the probability that genetic content of the population will be used to its fullest potential.

The convergence dynamics of an evolutionary algorithm are highly influential in affecting the quality of the evolved population. The effectiveness of these dynamics is highly inconsistent. This inconsistency is addressed by running the same problem dozens, if not hundreds of times, to derive statistical confidence in an assertion of the systems performance.

The cause of this variance lies in the stochastic component of the algorithm, which can sometimes promote sub-optimal solutions to a state of dominance in the population. As the sub-optimal genetic content is propagated throughout the population, relatively unfit but potentially useful genetic content is lost. This loss of genetic material eliminates the scope available to the algorithm for evaluating alternative solutions, thus resulting in evolutionary gridlock.

Even under circumstances wherein the dynamics of convergence are sustained by favourable random behaviour, there is a chance that the process of adaptation may become so acutely attuned to the irrelevant peculiarities of the training set

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 146–157, 2008.
© Springer-Verlag Berlin Heidelberg 2008

that the system loses generality, thus rendering it obsolete for the target application. Overfitting is particularly likely to occur in the later stages of evolution, when the algorithm has exhausted the most probable trajectories to higher fitness and must explore the possible improvements afforded by esoteric adaptations to the training signal.

These two phenomenon are known as premature convergence and overfitting respectively. Practitioners have attempted to address these principle pitfalls of GP by manipulating the dynamics of convergence in the algorithm. By improving the quality of convergence, the consistency of the algorithm to achieve its full potential is improved while the tendency of the algorithm to explore counter productive adaptations is inhibited.

The structure of the paper is as follows: A review of previous convergence manipulation techniques is given in Section 2 with a detailed description of Hereditary Repulsion provided in Section 3. An overview of the experimental setup is found in Section 4 followed by the three experiments of the paper. The first experiment, which demonstrates the strength of HR in contrast to a suite of benchmark systems is given in Section 5. The second experiment, in Section 6, evaluates the contribution of the repulsion tournament to the performance of GP. Section 7 confirms the results of Section 6 by incorporating the fitness constraint of HR into a steady state algorithm, resulting in drastically improved performance. Section 8 discusses the significance of the results along with future research directions.

2 Background

Manipulating convergence so as to improve the quality of evolution is a consistent theme of research in evolutionary algorithms. One of the first approaches to this problem was to inhibit dense concentrations of homogeneous solutions by forcing similar solutions to share their fitness [5]. *Fitness sharing* techniques thus attempt to emulate observations of natural biological systems wherein phenotypes exhibit niche specialisation. Many variants of this strategy have been pursued such as sequential niching, [2], speciation with implicit fitness sharing and co-evolution, [3] and a niching method [11] known as clearing.

While effective at encouraging the occupation of multiple local optimum, fitness sharing methods suffer from the need to set a priori parameters such as the similarity measure needed to define a minimum distance between optima. Such measures are hard to define and may need to change in the later stages of evolution, [11]. Hierarchal Fair Competition Model [7] has also demonstrated Using fitness as the primary segregation measure.

A highly effective convergence manipulation algorithm known as ALPs uses the concept of age to inhibit premature convergence [6]. The method uses age "bands" to define the population pools the individuals of the population may occupy. New individuals are continually introduced so as to maintain exploration. Normal evolutionary dynamics are present within the age "bands".

Finally, spatial segregation has been employed to prevent the spread of a single dominant individual. In this model, multiple initial populations are allowed to evolve, the idea being that each population will develop towards different optima. After a pre-specified number of evolutionary cycles, the "islands" [8] are allowed to intercommunicate. By such communication, the models are producing a number of beneficial effects.

Primarily, the homogenisation of each island is delayed with the introduction of such new material, with beneficial consequences for the ultimate convergence of the entire population of islands. Also, separate aspects of the final solution may be present in different islands, by communication these sub solutions are allowed to coexist and intermingle. The incompatibility of highly evolved solutions of the same domain but from different evolutionary runs is a well known phenomenon, an inter island communication protocol avoids this by maintaining coherence between the islands.

Simulated annealing [1] employs controlled randomness to reinvigorate the evolutionary search which also has a beneficial effect on the convergence dynamics.

3 Hereditary Repulsion

Hereditary Repulsion is a convergence manipulation protocol which can significantly improve the performance consistency of an evolutionary algorithm. HR was born out of the observation that the individuals in the later stages of evolution are descended from only a few individuals in the initial population [9].

This state, wherein the population is saturated with the progeny of a few dominant individuals, typically characterises premature convergence. As a pre-emptive measure against this eventuality, HR employs a "repulsion tournament" which inhibits crossover events between individuals with a similar hereditary history.

The algorithm begins by selecting an individual at random. This individual is used as the reference for the repulsion algorithm. A tournament pool of size N is then filled with random individuals. The shared hereditary history between the individuals in the pool and the reference individual is measured. The pool individual with the smallest hereditary overlap is selected to be crossed over with the reference individual. Figure 1 provides an example of how the overlap is calculated between two individuals. A HR tournament of size 1 is the same as random selection. This is because the reference individual is chosen randomly initially and since there is only 1 individual in the tournament who is also chosen randomly. The selection process is consequently equivalent to random selection, as the repulsion process has no other candidates against which to consider hereditary similarity.

Given that the repulsion tournament algorithm will place pressure on the dynamics to explore diverse representations, there is a possibility that the quality of the population will degrade as it explores the expression space. To protect the algorithm from the potential deleterious effects of intense exploratory dynamics,

a constraint was incorporated which mandated that an individual must be better than both its parents before it can be considered for insertion into the next generation.

It is notoriously difficult to accurately determine the effective diversity of a population due to well known incongruities between the semantic and syntactic content of a population. In Boolean problem domains, such as the Parity problem, one may map the binary sequence (of outputs from the set of standard inputs) associated with each solution to an integer and very accurately determine the phenotypic diversity of the population. Figure 2 contrasts the loss of phenotypic diversity between a HR system and a standard GP algorithm. These results are discussed in more detail in [10].

It is clear from Figure 2 that HR is extremely effective at sustaining diversity in the population. Significantly, HR does this without any explicit analysis of the population's content, in contrast to other techniques such as edit distance algorithms [4]. This facility broadens the applicability of HR to a wide range of evolutionary computation contexts.

Finally, due to the nature of the algorithm, HR will have a variable number of evaluations per generation. This is caused by the many rejected crossover events for each successful one. For this reason, the HR algorithm is provided with an upper limit to how many evaluations it may make during a run. For the sake of comparison, this upper limit is set equal to the total number of evaluations in a corresponding standard GP run.

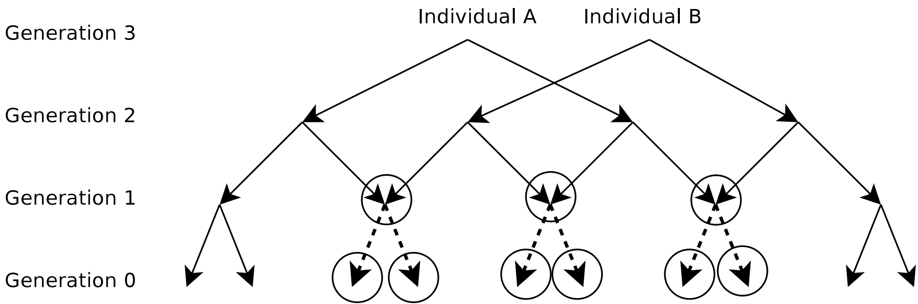


Fig. 1. Illustration of Common Hereditary History between 2 Individuals in a Generation System. In this example, individuals A and B share 9 common ancestors.

4 Experimental Overview

This section describes the experimental design used in deriving the results in this paper. The results in this paper demonstrate that the fitness constraint in HR consistently outperforms standard approaches with respect to both training and testing performance. Furthermore, we show that by incorporating this constraint into a standard steady state algorithm, a minor programmatical modification, one can expect similar performance increases. Such a claim requires extensive

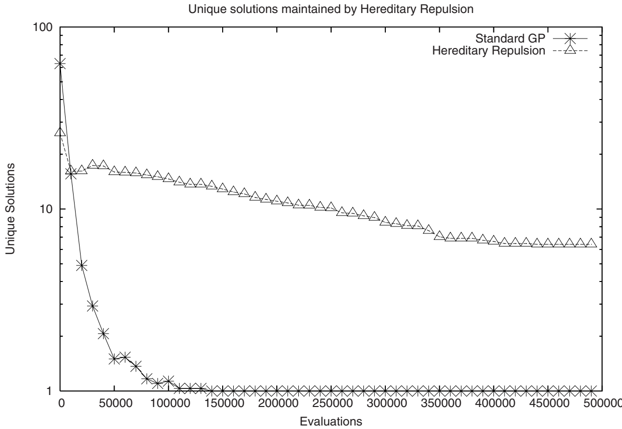


Fig. 2. Phenotypic Diversity on Parity 5

experimental evidence, and for this reason we demonstrate our results in a diverse range of problem domains including 3 classification problems and 3 regression problems. The parameters and primitives used for each setup are discussed in the following sections.

We employ a steady state and generational algorithm as benchmarks against which the performance of our algorithms are compared. For all experiments and all systems a population size of 100 was used and a tournament size of 2. The initial population of expressions was created using ramped half and half initialisation with a minimum initial depth of 2 and a maximum initial depth of 4. The generational system employed 2 elites to stabilise evolution. No mutation was used. All results shown are the average behaviour of 30 independent runs.

Due to space constraints, only testing results are shown, except for the Parity problem. The Parity problem is from the Boolean domain and thus has an finite set of inputs cases which can be enumerated in training, so the notion of testing performance is not relevant in this case. It must also be noted that testing performance is a better measure of system quality as it reflects generalisation, or learning the "true" distribution. This measure may decrease even when the system appears to be improving on the training data.

4.1 Binary Classification

Binary classification GP takes a set of input values and classifies the instance as being a positive or negative. The domains used were concerned with medical classification problems. These were the BUPA Liver Disorder problem, the Wisconsin Prognostic Breast Cancer problem and the Pima Indians Diabetes problem. These were all sourced from the UCI Machine Learning Database Repository.

A small number of incomplete records were discarded and the data sets were split into training and testing respectively. All input values were normalised by the associated maximum value for each column of input data. An output greater

than 0.5 was taken to be a positive result and an output less than 0.5 was taken to be a negative result.

The primitives used were {*,%,+,-}. Any divide by zero exception signalled the algorithm with an NAN result. Any such expressions were discarded by the algorithm. Evolution was executed for 500 generations or 50000 evaluations. The maximum tree depth was set to 8. Fitness measure was derived from the Root Mean of Squared Error (RMSE) : $1/(1 + RMSE)$.

4.2 Real Valued Regression

Real valued regression symbolically regresses functions in the real domain. The domains used were the Quartic Polynomial and the Rastrigin functions. The Rastrigin equation is a function of 2 variables, x and y. The range for both variables is $[-5.12, 5.12]$. The function is defined as : $f(x, y) = 20 + x^2 + y^2 - 10 * (cos2\pi x + cos2\pi y)$. The quartic polynomial is a function of 1 variable, x, in the range $[-1.0, 1.0]$. It is defined as $f(x) = x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x$. Both problems used 20 randomly sampled points for training and testing.

The primitives used were {*,%,+,- cos,sin}. Any divide by zero exception signalled the algorithm with an NAN result. Any such expressions were discarded by the algorithm. Evolution was executed for 500 generations or 50000 evaluations. The maximum tree depth was set to 8. Fitness measure was derived from the RMSE : $1/(1 + RMSE)$.

4.3 Odd 5 Parity

The Odd Parity problem takes a sequence of input bits and returns a 1 if the number of active bits is odd, a 0 otherwise. The number of input bits used was 5. This amounted to 2^5, 32 training cases.

The primitives used were {AND, OR, NOR, NAND}. This problem domain is notoriously difficult without the XOR or EQUAL functions, thus evolution was allowed to continue for 5000 generations or 500000 evaluations. The maximum tree depth was set to 16. Both the tree depth and period of evolution is much higher in the experiment as the problem domain is much more difficult. Fitness measure was the number of correct outputs (max 32).

5 Experiment 1: Validation of Method

This section presents the results of the experiments whose purpose was to validate the applicability of HR to a wide range of problem domains. The medical classification experiments, shown in Figures 3, 4 and 5 demonstrate HRs ability to resist overtraining and continuously improve the quality of the population well into the late stages of evolution. This is in stark contrast to the benchmark systems which despite being initialised with the same settings quickly degrade their ability to generalise.

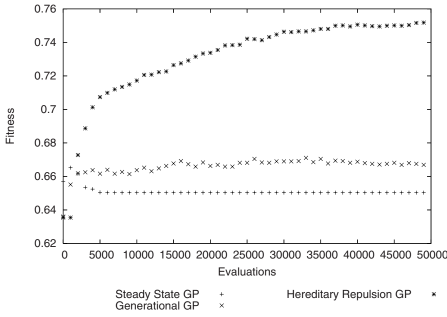


Fig. 3. Bupa Testing Results for Experiment 1

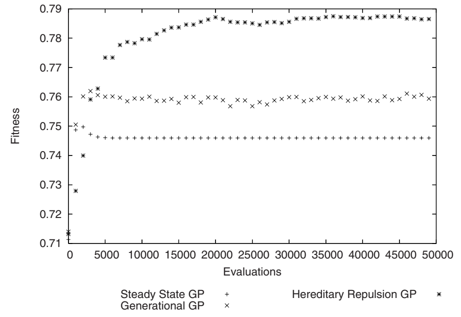


Fig. 4. Indian Testing Results for Experiment 1

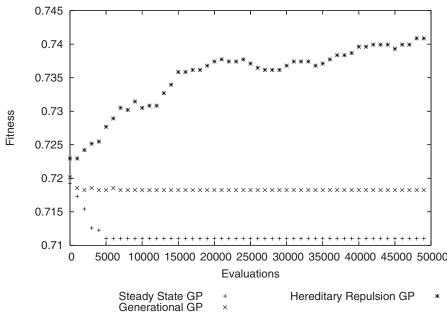


Fig. 5. WPBC Testing Results for Experiment 1

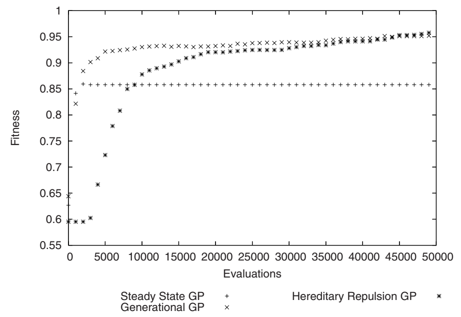


Fig. 6. Polynomial Testing Results for Experiment 1

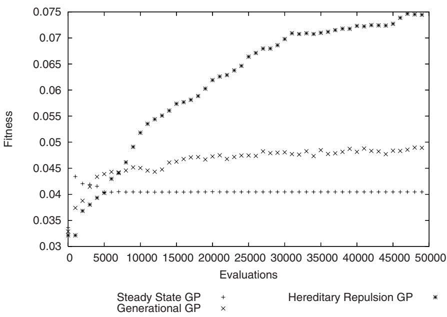


Fig. 7. Rastrigin Testing Results for Experiment 1

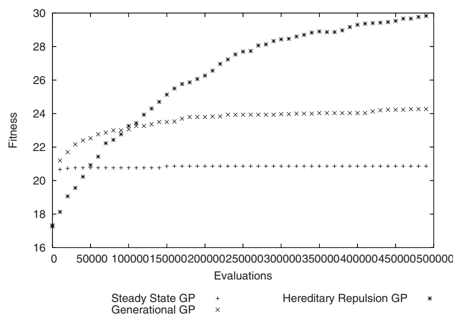


Fig. 8. Parity Training Results for Experiment 1

This behaviour is similar in the Rastrigin experiment on Figure 7 and the Polynomial experiment on Figure 6. A notable observation is that the generational GP system does quite well on the Polynomial experiment, however this is an exception across all the other problem domains. The Boolean regression results, in Figure 8 demonstrate the effectiveness of HR on the difficult Parity problem.

6 Experiment 2: Analysis

This experiment was designed to determine the contribution the repulsion algorithm makes to the performance of HR under normal circumstances. Results in [10] indicated that the repulsion algorithm was useful in extreme circumstances, such as when the population was very small.

The experiments used tournament sizes ranging from 1 to 5. A tournament size of 1 is equivalent to random selection, thus completely eliminating the effect of repulsion on the selection dynamics.

The results for all the problem domains, shown in Figures 9, 10, 11, 12, 13 and 14 do not indicate a clear difference between the different strategies. There are minor differences in some experiments, such as the slight degradation of the random selection HR system in the Rastrigin problem on Figure 13. Notably, these differences are not consistent across the problem domains and the differences between the approaches is minor.

This confirms earlier indications in [10] that the effect of the repulsion algorithm has a little bearing under circumstances where the population is relatively large. The main contribution to the increased performance is therefore the core constraint of HR. If this is truly the case, one would expect similar performance increases simply by including this constraint into the benchmark systems.

7 Experiment 3: Constraint Evaluation

This experiment demonstrates the immediate and tangible performance increases that can be expected simply by modifying an existing evolutionary algorithm to include the elementary constraint used by HR. To this end, we modified the benchmark steady state system with this constraint and evaluated its performance across the problem domains.

The change was completed by modifying a single line of code in the steady state system. Put simply; in our steady state system, if a child was to get into the population it had to be better than at least one of its parents. Our modification was that the child had to be better than both its parents. The steady state system used here also used random selection, rather than tournament or roulette wheel selection schemes. These were found to have a negative effect on performance.

The results for all the problem domains, shown in Figures 15, 16, 17, 18, 19 and 20 are a convincing demonstration of the effectiveness of the HR constraint.

While the constrained steady state GP system is more susceptible to the negative effects of highly evolved populations, such as overfitting, it dramatically outperforms the steady state benchmark system it was derived from on a range of diverse problem domains.

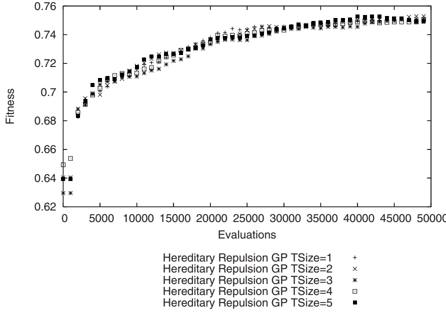


Fig. 9. Bupa Testing Results for Experiment 2

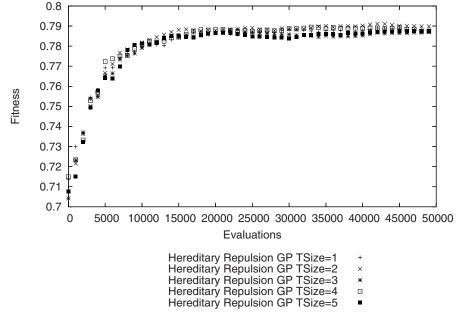


Fig. 10. Indian Testing Results for Experiment 2

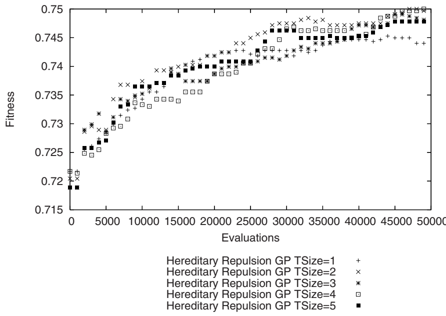


Fig. 11. WPBC Testing Results for Experiment 2

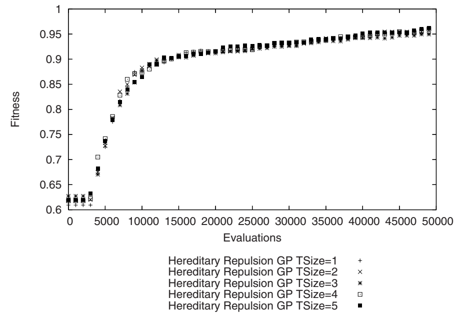


Fig. 12. Polynomial Testing Results for Experiment 2

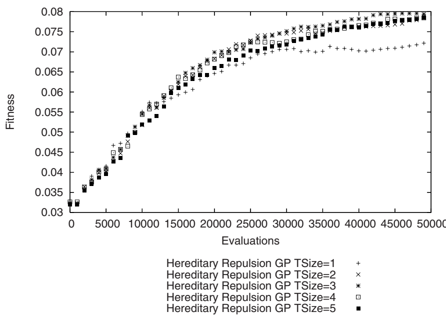


Fig. 13. Rastrigin Testing Results for Experiment 2

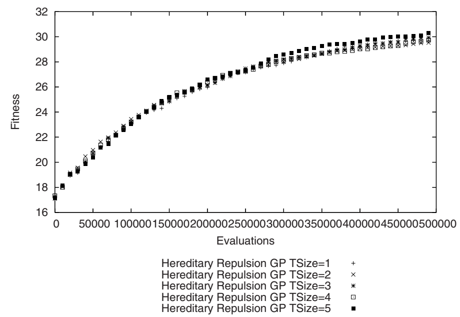


Fig. 14. Parity Training Results for Experiment 2

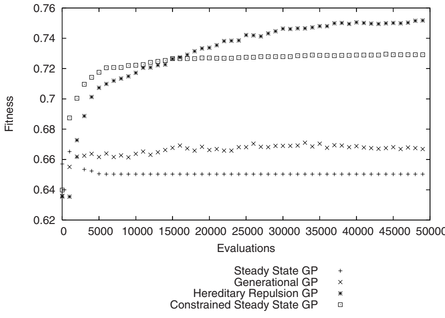


Fig. 15. Bupa Testing Results for Experiment 3

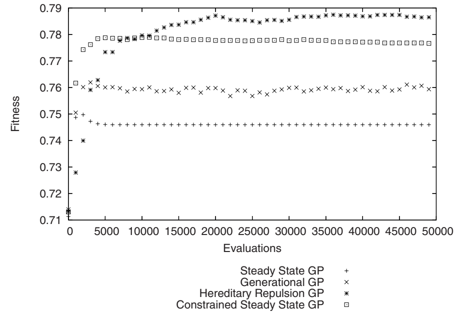


Fig. 16. Indian Testing Results for Experiment 3

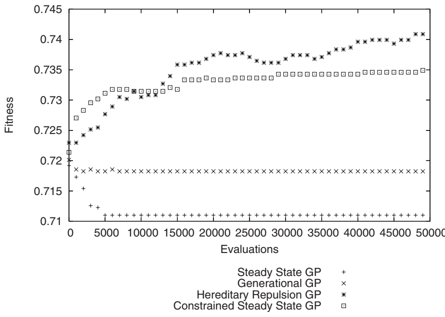


Fig. 17. WPBC Testing Results for Experiment 3

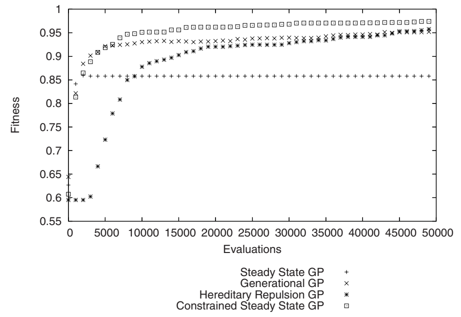


Fig. 18. Polynomial Testing Results for Experiment 3

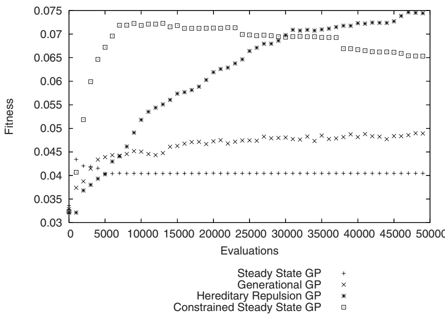


Fig. 19. Rastrigin Testing Results for Experiment 3

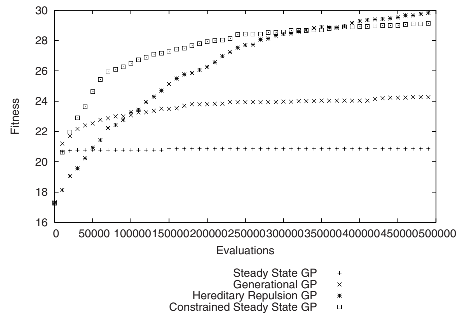


Fig. 20. Parity Training Results for Experiment 3

8 Discussion and Future Work

Hereditary Repulsion is a convergence manipulation protocol which can drastically improve the quality of evolution even under circumstances of extremely small populations. We have validated the ability of HR to improve evolution and significantly, resist the loss of generalisation associated with highly converged solutions.

Furthermore, we have shown how the essential simple constraint at the heart of the HR algorithm can be effortlessly incorporated into existing algorithms with immediate tangible improvements in the quality of the evolutionary process. Lacking the analytical tools required to prove the effectiveness of this approach, we have empirically demonstrated the power of this constraint in 6 different problem domains.

Why does the HR constraint result in such a dramatic improvement?. The original motivation behind the HR work was to eliminate the phenomenon of premature convergence. This is particularly likely when suboptimal solutions replace potentially useful but relatively unfit genetic material. By forcing the algorithm to improve against *both* its parents, the system has no choice but to progress through recombination events which combine genuinely useful aspects of both parents. This prevents the propagation of a single high quality individual while ensuring that useful content from both parents is propagated.

Future work will focus on methods to improve the efficiency of evolution. This may entail techniques to prevent repeated crossover events as well as extrapolation of potentially fruitful crossover sites. Another research focus will be the development of different convergence methodologies. Currently steady state and generational methods have strengths and weaknesses that are unique to each approach. Steady state preserves high quality solutions but suffers from overfitting while generational techniques are highly resistant to overfitting but suffer from the potential loss of high quality solutions over time. Perhaps by incorporating aspects of both algorithms, it may be possible to reap their combined benefits.

References

1. Cordón, O., de Moya Anegón, F., Zarco, C.: A new evolutionary algorithm combining simulated annealing and genetic programming for relevance feedback in fuzzy information retrieval systems. Soft. Comput 6(5), 308–319 (2002)
2. Beasley, D.R.B.D., Martin, R.R.: A sequential niche technique for multimodal function optimization. In: Evoluionary Computation, vol. 1, pp. 101–125 (1993)
3. Darwen, P., Yao, X.: Speciation as automatic categorical modularization. In: IEEE Transactions on Evolutionary Computation, vol. 1, IEEE Computational Intelligence Society, Los Alamitos (1997)
4. Edmund, S.G., Burke, K., Hendall, G.: Diversity in genetic programming: An analysis of measures and correlation with fitness. In: IEEE Transactions on Evolutionary Computation, vol. 8, IEEE Computational Intelligence Society, Los Alamitos (2004)
5. Goldberg, D.E., Richardson, J.: Genetic algorithms with sharing for multimodal function optimisation. In: Proc. 2nd Int Conf. Genetic Algorithms, pp. 41–49 (1987)

6. Hornby, G.S.: Alps: the age-layered population structure for reducing the problem of premature convergence. Gecco 2005 (2005)
7. Hu, J., Goodman, E.D.: The hierarchical fair competition (HFC) model for parallel evolutionary algorithms. In: Fogel, D.B., El-Sharkawi, M.A., Yao, X., Greenwood, G., Iba, H., Marrow, P., Shackleton, M. (eds.) Proceedings of the 2002 Congress on Evolutionary Computation CEC 2002, pp. 49–54. IEEE Press, Los Alamitos (2002)
8. D. J. S. M. X.-Y. L. R. M., C. J.: Automated detection of nodules in the ct lung images using multi-modal genetic algorithm. In: IEEE Transactions on Evolutionary Computation, vol. 1, IEEE Computational Intelligence Society, Los Alamitos (2003)
9. McPhee, N., Hopper, N.: Analysis of genetic diversity through population history. In: Gecco, pp. 1112–1120 (1999)
10. Murphy, G., Ryan, C.: Manipulation of convergence in evolutionary systems. In: Genetic Programming Theory and Practise, Springer, Heidelberg (2007)
11. Sareni, B., Krahenbuhl, L.: Fitness sharing and niching methods revisited. In: IEEE Transactions on Evolutionary Computation, vol. 2, IEEE Computational Intelligence Society, Los Alamitos (1998)

Crossover, Sampling, Bloat and the Harmful Effects of Size Limits

Stephen Dignum and Riccardo Poli

Department of Computing and Electronic Systems,
University of Essex,
Wivenhoe Park, Colchester, CO4 3SQ, UK
{sandig,rpoli}@essex.ac.uk

Abstract. Recent research [9,2] has enabled the accurate prediction of the limiting distribution of tree sizes for Genetic Programming with standard sub-tree swapping crossover when GP is applied to a flat fitness landscape. In that work, however, tree sizes are measured in terms of number of internal nodes. While the relationship between internal nodes and length is one-to-one for the case of *a*-ary trees, it is much more complex in the case of mixed arities. So, practically the *length* bias of subtree crossover remains unknown. This paper starts to fill this theoretical gap, by providing accurate estimates of the limiting distribution of lengths approached by tree-based GP with standard crossover in the absence of selection pressure. The resulting models confirm that short programs can be expected to be heavily resampled. Empirical validation shows that this is indeed the case. We also study empirically how the situation is modified by the application of program length limits. Surprisingly, the introduction of such limits further exacerbates the effect. However, this has more profound consequences than one might imagine at first. We analyse these consequences and predict that, in the presence of fitness, size limits may initially speed up bloat, almost completely defeating their original purpose (combating bloat). Indeed, experiments confirm that this is the case for the first 10 or 15 generations. This leads us to suggest a better way of using size limits. Finally, this paper proposes a novel technique to counteract bloat, sampling parsimony, the application of a penalty to resampling.

Keywords: Genetic Programming, Theory, Crossover, Search, Sampling, Bloat, Program Length.

1 Introduction

With the advent of a greater understanding of program search spaces—for example we now know that the functionality of programs reaches a limit as program length increases [6,4,5]—acquiring knowledge on how GP operators sample program length classes has become more and more urgent. Ideally, we would like to sample the length class where the smallest optimal programs can be found. Unfortunately, in general: a) one does not know where solutions (let alone most compact ones) are, and, b) genetic operators present specific length biases which are often unknown or only partially known and, therefore, are difficult to direct and control. In any case, a characterisation of operator bias is needed in understanding how GP will sample the search space in the first instance before technically sound problem specific modifications can be made.

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 158–169, 2008.
© Springer-Verlag Berlin Heidelberg 2008

GP, of course, applies a number of competing operators that like to sample the search space in different ways. In this paper we look at the application of standard crossover with uniform selection of crossover points, an operator for which recent research [9,2] has enabled the accurate prediction of the limiting distribution in the absence of selection, i.e., when GP is applied to a flat fitness landscape. In that work, however, tree sizes are measured in terms of number of internal nodes, which is not what GP users normally want and use. While the relationship between internal nodes and length is one-to-one for the case of a-ary trees, it is much more complex in the case of mixed arities. So, practically the *length* bias of subtree crossover remains unknown.

This paper starts filling this theoretical gap, by extending previous research [9,2] to include terminals as well as internal nodes in our program length distribution (Section 2). This shows that crossover will sample increasingly more smaller programs as the distribution converges. As smaller programs are less numerous than larger ones a large amount of resampling takes place. Empirical evidence gathered using two standard GP benchmark problems confirms this bias (Section 3). Although, selection is likely to initially work against the biases of crossover, as fitness converges, either during the later stages of a GP run or if an area of neutrality is reached, this bias will become more acute. In Section 4 we also study empirically how the situation is modified by the application of program length limits. Unexpectedly, the changes to the sampling biases introduced by such limits, further exacerbate bloat in early generations, thereby reducing the efficacy of size limits as mechanisms for bloat control.

In [2] a theory was put forward, the Crossover Bias bloat theory, which postulates that the main reason for bloat is precisely the oversampling of short programs produced by subtree crossover. Our findings confirm this tendency with and without length limits. So, in Section 5 we propose a novel technique to indirectly counteract bloat, sampling parsimony, which is effectively the application of a penalty for resampling. We study its behaviour with and without fitness. In both cases, we show that applying even slight resampling penalties, program growth can significantly be sped up or slowed down depending on the application of the penalty. Applying a 'resampled' penalty to programs confirms the crossover bias bloat theory of [2]. Applying a newly sampled program penalty, provides a natural way of acting on the very roots of bloat: the sampling and re-sampling of short programs.

We draw our conclusions in Section 6.

2 Program Length Distributions in GP

2.1 Internal Node Distributions

In [9] strong theoretical and experimental evidence was provided that standard sub-tree swapping crossover with uniform selection of crossover points pushes a population of a-ary GP trees towards a limiting distribution of tree sizes of the form:

$$\Pr\{n\} = (1 - ap_a)\binom{an+1}{n}(1 - p_a)^{(a-1)n+1}p_a^n \quad (1)$$

This is known as a Lagrange distribution of the second kind. $\Pr\{n\}$ is the probability of selecting a tree with n internal nodes and a is the arity of functions that can be used in

the creation an individual. The parameter p_a was shown to be related to μ_0, the mean program size at generation 0, and a according to the formula:

$$p_a = \frac{2\mu_0 + (a-1) - \sqrt{((1-a)-2\mu_0)^2 + 4(1-\mu_0^2)}}{2a(1+\mu_0)} \quad (2)$$

Equation (1) was generalised using the Gamma function: $\Gamma(n+1) = n!$ in [2] to enable mixed arity tree internal node distributions to be predicted:

$$\Pr_g\{n\} = (1 - \bar{a}p_{\bar{a}}) \frac{\Gamma(\bar{a}n+2)}{\Gamma((\bar{a}-1)n+2)\Gamma(n+1)} (1-p_{\bar{a}})^{(\bar{a}-1)n+1} p_{\bar{a}}^n \quad (3)$$

$\bar{a}$ being an averaged arity of the primitive set. This can be calculated for mixed function arities from experimental initialisation parameters as follows:

$$\bar{a} = E(arity(F)) = \sum_f arity(f)P(F=f) \quad (4)$$

where f is a non-terminal to be used in the GP experiment, $arity(f)$ is a function returning the arity of the non-terminal f, and $P(F = f)$ is the probability that a particular non-terminal f will be selected for a non-terminal node by the tree initialisation procedure. For traditional FULL and GROW initialisation methods non-terminals are chosen with equal probability [7].

2.2 Program Length Distributions

Our first step towards extending Equation (3) to allow us to predict program length distributions with mixed arities, is to look at what we can say for certain regarding the relationship between number of internal nodes and program length. First, we know for a-ary trees where the arities of all nodes in the tree are the same, the length, ℓ, of a program can be expressed exactly in terms of the number of its internal nodes, n, using the following equation

$$\ell = a \times n + 1, \quad (5)$$

where a is the (fixed) arity of the internal nodes. Therefore, rearranging Equation (5) to obtain internal nodes in terms of length, i.e.,

$$n = \frac{\ell - 1}{a}, \quad (6)$$

and substituting this into Equation (1), we obtain that, for a-ary trees,

$$\Pr_l\{\ell\} = \begin{cases} \Pr\{\frac{\ell-1}{a}\} & \text{if } \ell \text{ is a valid length (i.e., } \frac{\ell-1}{a} \text{ is a non-negative integer),} \\ 0 & \text{otherwise,} \end{cases} \quad (7)$$

where $\Pr_l\{\ell\}$ is the limiting distribution of program lengths. This distribution applies, for example, for Boolean function induction problems where often all functions are binary and symbolic regression problems where often only the standard four arithmetic operations are used.

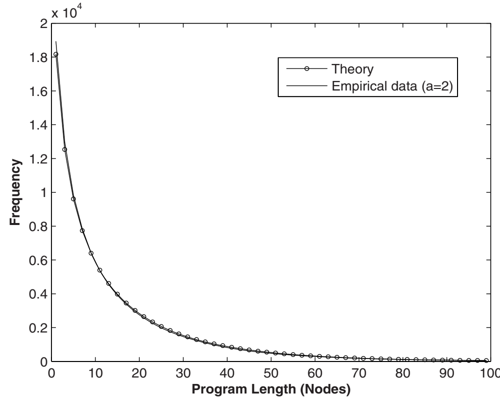


Fig. 1. Comparison between theoretical and empirical program length distributions for 2-ary trees initialised with FULL method (depth=3, initial mean size $\mu_0 = 15.0$, mean size after 500 generations $\mu_{500} = 14.49$). Invalid even lengths are ignored.

Figure 1 shows an observed plotted length distribution for 2-ary trees, with invalid (even) lengths removed, compared to that predicted by $\Pr_l$. The observed values are averages over twenty independent runs with populations of 100,000 individuals run for 500 generations.[1] As we can see there is a very close fit between the two curves.

Our next step is to extend the generalised formula for mixed-arity trees (Equation (3)) so as to predict length distributions rather than internal node distributions. We know that for a program length of 1, a single terminal, there will always be 0 internal nodes. Therefore, the predicting single node programs is a simple one-to-one mapping with the generalised formula for 0 internal nodes. However, other lengths can be obtained by different combinations of internal nodes of different arities. For example, one can obtain programs of length 3 by using one internal node of arity 2 or two internal nodes of arity 1.

As a first approximation, we will assume that we can still estimate the expected number of internal nodes in a tree of length ℓ by applying Equation (6), simply using $\bar{a}$ instead of a. We can then substitute the result into Equation (3) to obtain the distribution of lengths we are looking for. Naturally, between the variable ℓ and the variable n there is a difference in scale (the factor $\bar{a}$). So, we will need to normalise the values produced by Equation (3) to ensure the new distribution sums to 1.

Putting all of this together, we obtain an approximate model of the limiting distribution of program lengths in the case of primitive sets of mixed arities. Namely:

$$\Pr_v\{\ell\} = \begin{cases} \Pr_g\{0\} & \text{if } \ell = 1, \\ \frac{\Pr_g\{\frac{\ell-1}{\bar{a}}\}}{\bar{a}} & \text{if } \ell \text{ is a valid length greater than 1.} \end{cases} \tag{8}$$

Note, we do not require $\frac{\ell-1}{\bar{a}}$ to be an integer.

Since there were approximations in the original derivation of Equation (3) in [2], and we added further approximations in the derivation of Equation (8), one might wonder whether the model is sufficiently accurate to be of practical use. Figure 2 shows

[1] These and all other experimental parameters were chosen as in [2] for ease of comparison.

observed and theoretical values of the limiting length distribution experiment set up for internal nodes of arities of 1 and 2 where all lengths are valid, whilst Figure 3 compares the theoretical and empirical distribution obtained in a GP run with the primitive set of the Artificial Ant problem, which has internal nodes arities of 2, 2 and 3, for IF-FOOD-AHEAD, PROGN2 and PROGN3, respectively. Note, with this choice there is no way of generating programs of length $\ell = 2$. Finally, Figure 4 shows the results of using arities of 1, 2, 3 and 4. Note that in order to highlight the fit for larger and less common programs we used a log scale for frequency.

As one can see, the model in Equation (8) accurately models the distribution observed in real runs in all cases, with only minor deviations at the very short program lengths where some of the assumptions behind the model are less applicable.[2] However, generally both the theoretical model and the actual runs show that in almost all cases crossover will sample with high frequency small programs. The effects of this bias are investigated in the next section.

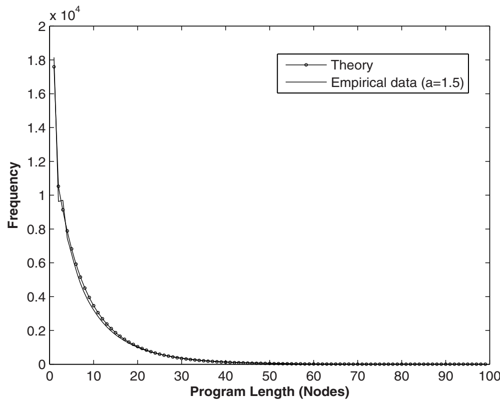


Fig. 2. Comparison between theoretical and empirical program length distributions for trees created with arity 1 and 2 functions initialised with FULL method (depth=3, initial mean size μ_0 = 8.13, mean size after 500 generations μ_{500} = 8.51). All lengths are valid.

3 Sampling and Resampling

Our first step is to see how standard crossover will sample the search space on a flat fitness landscape. Our primary purpose for doing this is simply to isolate the search bias for crossover. It should be noted, however, that, while in the presence of fitness gradients selection will counteract the crossover bias (this is analysed further in conjunction with selection in Section 5), there are situations where the crossover bias may become the prominent search bias. This may happen, for example when GP search reaches an area of neutrality, e.g., when GP operators, during an experimental run, are unable to escape areas of similar fitness.

[2] Curing the slight mismatches for earlier lengths would require a more accurate estimation of number of internal nodes of each arity for small ℓ. We will investigate more precise models in future work.

In particular we are interested in finding out how much resampling goes on. This gives us an idea of the efficiency or otherwise of the search. To empirically analyse crossover sampling we took two out-of-the-box problems from the ECJ evolutionary toolkit [8]: 4 Bit Even Parity and the Artificial Ant. As the Parity problem uses Boolean operators only we know that, in the absence of selection, the limit program length distribution to be that of a 2-ary tree as shown in Figure 1, whilst, as previously discussed, the Artificial Ant will follow a distribution similar to the one in Figure 3.

Adjustments were made to ECJ to remove mutation, ensure uniform selection of crossover points, and to prevent a depth limit being applied. A population size of 1,000 individuals was used and the results for 200 generations were averaged over one hundred independent runs. All experiments were initialised using the RAMPED method [3] with a maximum depth of 6 and minimum depth of 2. A constant fitness value was returned in all cases.

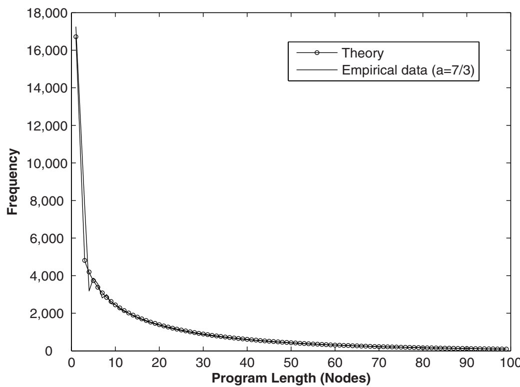


Fig. 3. As in Figure 2 but for arities 2, 2 and 3 (μ_0 = 32.12, μ_{500} = 33.22). Invalid length 2 is ignored.

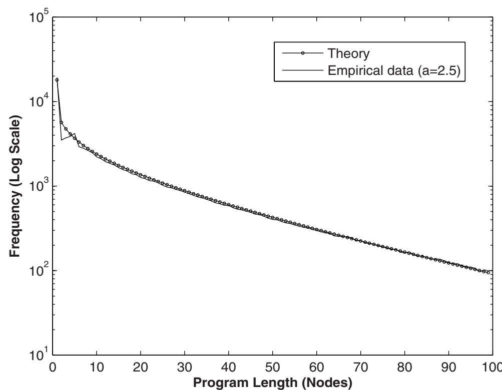


Fig. 4. As in Figure 2 but for arities 1, 2, 3 and 4 (μ_0 = 25.38, μ_{500} = 23.76). All lengths are valid.

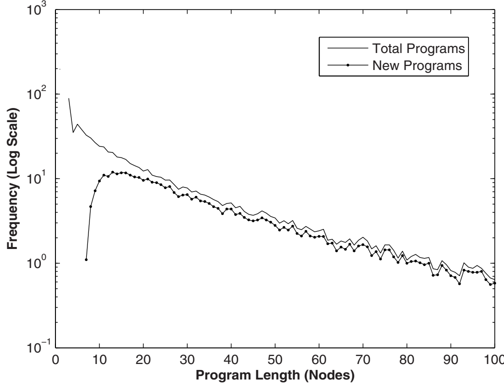


Fig. 5. Frequencies of new unique programs not sampled previously compared to all programs generated at generation 200, for the Artificial Ant Problem applied to a flat fitness landscape

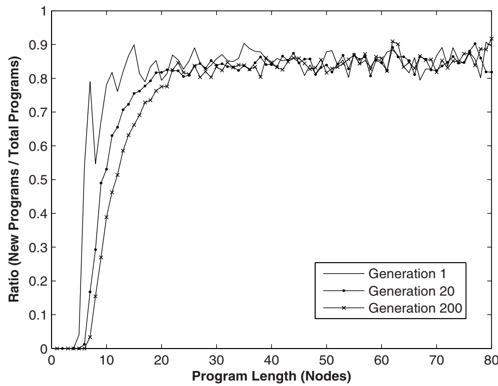


Fig. 6. Ratio of new unique programs not sampled previously compared to programs generated at generations 1, 20 and 200, for the Artificial Ant Problem applied to a flat fitness landscape

The total number of programs for each length was recorded at each generation along with the number of programs for each length that had been sampled in a previous generation. Taking the artificial ant problem, as we can see in Figure 5, at generation 200 the number of new unique programs is extremely small compared to that of the total for that generation. The majority of all programs sampled under these conditions are of course in the smaller length classes.

As a ratio, new programs divided by total programs, plotted in Figure 6, it is clear that newly sampled programs are being generated at the larger length classes and that crossover is progressively resampling more and more programs.

4 Effects of Size Limits

The standard technique to control bloat, namely the application of a depth or length limit, is known to have significant effects on GP dynamics (see, for example, [1]). Unfortunately, we don't have a mathematical model for the limit distribution of sizes (neither in terms of internal nodes nor in terms of lengths) in the presence of length limits. However, we can conduct experimentation to study their effects on such a distribution. Figure 7 shows the affect of applying length limits of 25, 50 and 100 to the Artifical Ant problem. The effect of the length limit is that programs become more frequent in the smaller length classes. This over-sampling exacerbates the wasteful resampling of programs of smaller lengths.

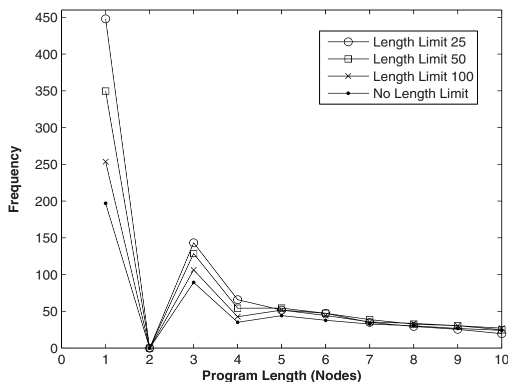


Fig. 7. Comparison of sampling frequencies associated with length limits for the Artificial Ant Problem applied to a flat fitness landscape

In the presence of fitness, this effect can be counteracted but not cancelled by selection. So, one should expect more sampling and resampling of short programs. However, following the line of reasoning of the crossover bias bloat theory [2], we know that for most problems these programs cannot be solutions, and in fact are typically very unfit, and, so, longer programs will be preferentially selected, leading to bloat. Thus, *size limits effectively increase the tendency to bloat since they induce more sampling of short programs, and, so, in the presence of non-flat fitness landscapes, GP populations rush towards the limit even more quickly than in the absence of the size limit!* This effect is particularly clear if one looks at the mode (the peak) of the program length distribution with and without length limits. Figure 8 shows how the mode (averaged over 100 independent runs) changes generation by generation for different limits in the case of the Parity problem (with selection). We can see that smaller size limits encourage GP to sample larger programs in the early generations before the size limit is reached. We found this effect because we looked into how the crossover sampling bias interacts with size limits. The effect has never been noticed before, probably because it becomes apparent only if one uses the right statistical tools: the mode of the size distribution (which is almost never used in reporting GP results).

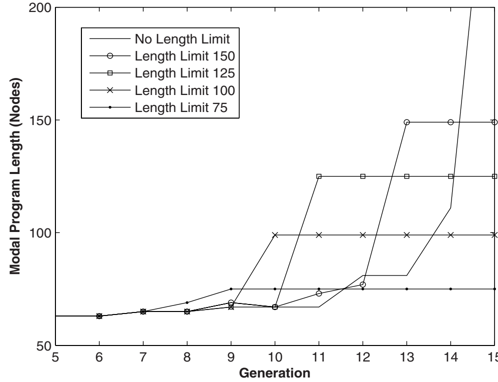


Fig. 8. Comparison of modal (peak) classes associated with length limits for the 4 Bit Even Parity Problem with selection

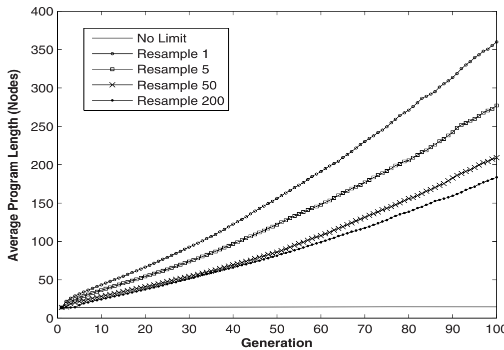


Fig. 9. Comparison of average program size applying resampling limits to the 4 Bit Even Parity Problem with a flat fitness landscape

These results suggest that, if size limits are imposed to combat bloat, then these should not be applied from generation 1, but much later and on demand, for example, if the average program size exceeds some pre-fixed threshold. This would avoid speeding up program growth in the early generations of a run.

Naturally, virtually all methods to combat bloat give more selective preference to shorter program than to longer ones. If in so doing they cause an oversampling of the short programs w.r.t. the base case (i.e., in the absence of the anti-bloat method)—which many do—then we should expect this phenomenon to still take place also with other bloat-control mechanisms, although perhaps with a lesser degree. We will explore this issue in future research.

5 Bloat and Sampling Parsimony

In section 3 we looked at how crossover likes to progressively sample smaller programs and the resulting resampling of programs, hence re-evaluations, that result from this. In this section we look at the prevention of resampling and its effect on program length.

To understand the effect of resampling and to control it, we have employed a novel technique which we have called *Sampling Parsimony*. This has two parameters, a resampling penalty to be applied, which is implemented as a percentage reduction of fitness, and a count of the number of times that a unique program can be sampled before that penalty is applied or removed.

Our first application is to look at how average program length will be affected by the application of a super penalty ensuring that a resampled program will not be reselected in the next generation. Using our standard ECJ problems with parameters as described in section 3 from Figure 9 we can see that, as we progressively prevent resampling by lowering our resampling limit, we increase the average size of the programs in our population. We have in effect created an effective fitness landscape [6] where the ability for a child to exist in the next generation is solely determined by whether that program has previously been sampled.

From our earlier analysis it, is unsurprising that we see that by depressing the fitness of resamples we will increase the sampling of larger programs, thereby increasing the average program size as we are in effect penalising smaller programs. What is more interesting is that we have managed to isolate the Crossover Bias bloating effect as described in [2]. Our method only penalises children and prevents them from being parents rather than preventing their creation. GP, therefore, uses larger programs as parents (see Figure 6), hence, increasing the average size of children and thereby increasing the average program size in the next generation. As smaller children are still created by crossover but have no chance of being chosen by selection, this process will continue. Even a relatively large resampling allowance of 200 on our flat landscape will greatly increase program size.

We apply our resampling penalty method to the Ant Problem with selection in Figure 10. We can see that our penalty, increases program growth within 100 generations. This is because we have, effectively, accelerated the Crossover Bias effect (crossover creating small programs that selection then ignores) already present in the 'No Limit' distribution. Practically, we can see that this acceleration only happens beyond a problem specific value of the number of resamples allowed, suggesting that experimental resampling restrictions may not attract significant additional program growth once an acceptable limit has been determined.[3]

Finally we reverse our method to apply a penalty to all programs from the beginning. We only remove the penalty after a specific number of resamples have been achieved, thereby allowing a program to be selected as a parent only after it has been sampled a number of times. From Figure 11 we can see that program growth is significantly reduced by applying a single sample penalty, whilst progressively increasing the sampling threshold before normal fitness is applied will reduce program growth towards a limit of approximately 50 samples.[4]

[3] Experimentation showed that no changes are observed beyond 5 resamples for the 4 Bit Even Parity problem, and approximately 15+ for the Artificial Ant.

[4] Approximately 5 for the Parity Problem, again the threshold is problem dependant.

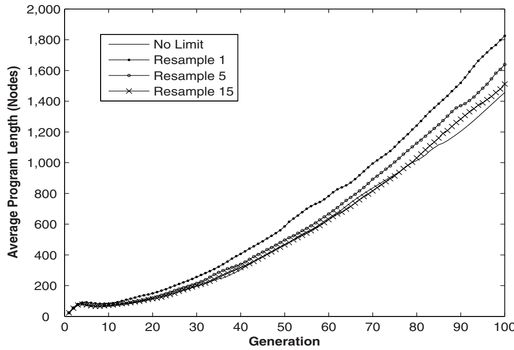


Fig. 10. Comparison of average program size applying resampling limits to the Artificial Ant Problem with selection

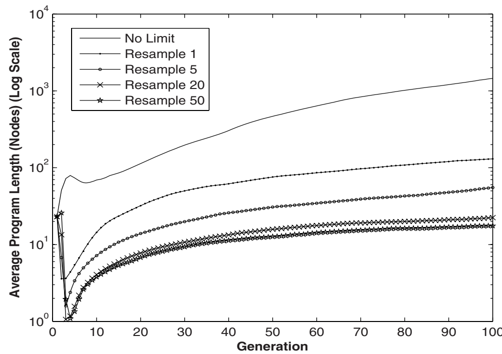


Fig. 11. Comparison of average program size applying sampling penalties to the Artificial Ant Problem with selection

Although its effect on bloat is self evident, it remains to be seen whether the sampling parsimony method can be successfully applied to improving overall program fitness over an entire run. We leave this for future work. The current 'blanket' method is of course very unsophisticated in that we prevent entire search spaces from being investigated without regard to program fitness. However, we believe that this remains an interesting technique that is worth exploring in greater depth and which might find application is a variety of areas, including, for example, escaping experimental stagnation under various conditions.

6 Conclusions

In this paper we have presented a limiting length distribution for GP with standard crossover with uniform selection of crossover points. This distribution now includes external nodes along with internal nodes, thereby extending previous research. Empirical validation confirms the accuracy of our model. Both theory and experiments show that the application of this form of crossover will quickly enable a population to converge to

a distribution that will exponentially sample smaller programs. As there are exponentially fewer unique smaller programs than larger ones, the sampling of new programs becomes less likely during a GP run if only crossover is applied. The effect becomes more prevalent as fitness values converge. This bias also becomes more acute with the application of a length limit, where, in addition to wasting more resources in resamples, it has further important consequences. In particular, we find that size limits initially speed up bloat, almost completely defeating their original purpose of combating bloat.

Although the application of selection before any fitness convergence will work against the crossover bias, smaller programs will always be created by crossover. As it is unlikely that these programs will be able to obtain a reasonable fitness, particularly during later stages of a GP run, they will be ignored by selection for the next generation and only larger parents will be selected. The continuing application of selection and crossover, therfore, causes the mean program size to increase, thereby creating bloat.

To explore what happens if one directly addresses this sampling-related cause for bloat, we have introduced a novel technique called Sampling Parsimony to tackle bloat. Curiously, this can be used accelerate growth as well as to reduce its effect. We have not, however, directly verified if Selection Parsimony is competitive with other anti-bloat techniques. We will address this question in future work.

References

1. Crane, E.F., McPhee, N.F.: The effects of size and depth limits on tree based genetic programming. In: Yu, T., Riolo, R.L., Worzel, B. (eds.) Genetic Programming Theory and Practice III, Ann Arbor, May 12-14. Genetic Programming, ch. 9, pp. 223–240. Springer, Heidelberg (2005)
2. Dignum, S., Poli, R.: Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In: Thierens, D., Beyer, H.-G., Bongard, J., Branke, J., Clark, J.A., Cliff, D., Congdon, C.B., Deb, K., Doerr, B., Kovacs, T., Kumar, S., Miller, J.F., Moore, J., Neumann, F., Pelikan, M., Poli, R., Sastry, K., Stanley, K.O., Stutzle, T., Watson, R.A., Wegener, I. (eds.) GECCO 2007: Proceedings of the 9th annual conference on Genetic and evolutionary computation, London, July 7-11, vol. 2, pp. 1588–1595. ACM Press, New York (2007)
3. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
4. Langdon, W.B.: How many good programs are there? How long are they? In: De Jong, K.A., Poli, R., Rowe, J.E. (eds.) Foundations of Genetic Algorithms VII, Torremolinos, Spain, Sepember 4-6 2002, pp. 183–202. Morgan Kaufmann, San Francisco (published, 2003)
5. Langdon, W.B.: Convergence of program fitness landscapes. In: Cantú-Paz, E., Foster, J.A., Deb, K., Davis, L., Roy, R., O'Reilly, U.-M., Beyer, H.-G., Kendall, G., Wilson, S.W., Harman, M., Wegener, J., Dasgupta, D., Potter, M.A., Schultz, A., Dowsland, K.A., Jonoska, N., Miller, J., Standish, R.K. (eds.) GECCO 2003. LNCS, vol. 2724, pp. 1702–1714. Springer, Heidelberg (2003)
6. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer, Heidelberg (2002)
7. Luke, S.: Two fast tree-creation algorithms for genetic programming. IEEE Transactions on Evolutionary Computation 4(3), 274–283 (2000)
8. Luke, S.: ECJ 13: A Java-based Evolutionary Computation Research System (2005), http://cs.gmu.edu/~eclab/projects/ecj/
9. Poli, R., Langdon, W.B., Dignum, S.: On the limiting distribution of program sizes in tree-based genetic programming. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 193–204. Springer, Heidelberg (2007)

The Performance of a Selection Architecture for Genetic Programming

David Jackson

Dept. of Computer Science, University of Liverpool
Liverpool L69 3BX, United Kingdom
djackson@liverpool.ac.uk

Abstract. Hierarchical decomposition and reuse techniques are seen as making a vital contribution to the scalability of genetic programming systems. Existing techniques either try to identify and encapsulate useful code fragments as they evolve, or else they rely on intelligent prior deconstruction of the problem at hand. The alternative we propose is to base decomposition on a partitioning of the input test cases into subsets or ranges. To effect this, the program architecture of individuals is such that each subset is dealt with in an independently evolved branch, rooted at a selection node handling branch activation. Experimentation reveals that performance of systems employing this architecture is significantly better than that of more conventional systems.

1 Introduction

Although genetic programming (GP) has proved itself a valuable tool for finding solutions to many and varied problems, the complexities of real-world domains make it vital that methods be found for scaling up GP techniques to solve larger-scale, more difficult tasks. Since the typical approach taken by human software engineers is to decompose a problem into simpler, more manageable sub-tasks, it seems natural that analogous approaches should be investigated as a means for dealing with complexity in genetic programming.

To this end, many researchers have extended and adapted GP systems so that instead of being monolithic or single-level, they accommodate decomposition and reuse mechanisms that allow evolution to proceed in a more hierarchical manner. The best established of these adaptations is that of Koza's Automatically Defined Functions (ADFs) [1-3], in which a number of function-defining branches are evolved alongside a main branch that can invoke and pass arguments to those functions. Other approaches include the Module Acquisition technique of Angeline and Pollack [4,5], Rosca and Ballard's Adaptative Representation through Learning (ARL) algorithm [6], and a two-stage method proposed by Roberts et al [7]. The importance of modularisation has also been recognised for genetic programming systems which use representations that differ from the tree structure usually employed. In his work on Cartesian Genetic Programming (CGP), for example, Miller has described bottom-up techniques for building hardware circuits from simpler 'cells' [8], and for encapsulation of modules in evolving CGP programs [9,10].

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 170–181, 2008.
© Springer-Verlag Berlin Heidelberg 2008

In many of the hierarchical systems that have been described in the literature, the focus is on attempting to identify and then encapsulate useful fragments of code as they appear during the evolutionary process. Typically, the number and nature of such fragments is not known or specified in advance, and so program architectures are often decided upon in a fairly ad-hoc way. However, in direct contrast to this, there have been some attempts to tackle problems by initially evolving solutions to tasks which are precisely defined, and which can contribute genetic material which is useful in a higher-level evolutionary process addressing the original problem [11-15].

In this vein, we propose the use of a GP program architecture which is also hierarchical in nature and which also has a form dictated by a pre-specified decomposition of the problem. The distinguishing characteristic of this architecture is that it is based not upon any intelligent insight into the nature of program behaviour and structure, but rather upon a fairly mechanical breakdown of the program's inputs into subsets. In the next section, we provide more details of this selection architecture, and then go on to examine its performance in comparison with other GP approaches.

2 A Selection Architecture

In most conventional GP systems, the program code of individuals is represented in a tree structure, the internal nodes being members of the function set, and the leaf nodes being taken from the terminal set. Alternatives to this format include linear code and more general graph structures. Attempts to evolve code that is more hierarchical in nature, and which makes use of evolving sub-structures, involve modifications of these basic forms.

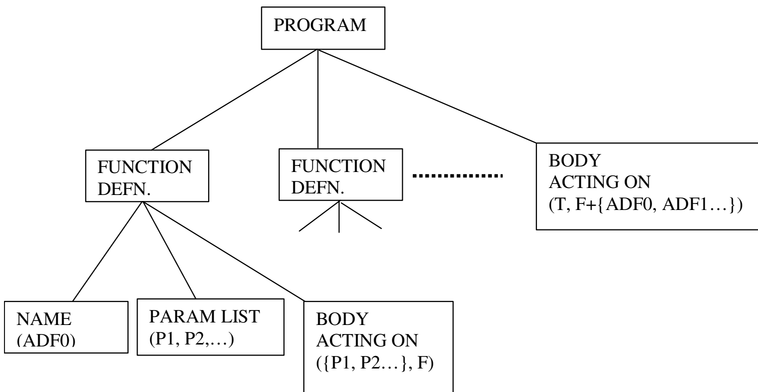


Fig. 1. Typical program architecture for ADF-based GP systems

The most well-known of the hierarchical approaches to GP is that of Koza's Automatically Defined Functions (ADFs). Typically, the architecture of programs in an ADF-based system is as shown in Figure 1. In this, a program tree comprises one or more function-defining branches and a main branch that may invoke those functions. Each function definition has a formal parameter list, and only members of

this list can appear as terminals in the body of the function; the terminal set of the problem itself is not accessible within an ADF. The body of the program as a whole – its main branch – is built from the usual terminal set, members of the problem function set, and members of the set of ADFs. During evolution, operators are usually subject to some context restrictions; in crossover, for example, if the first parent's crossover point is within, say, ADF0, then the second parent's crossover point is also confined to ADF0. In this way, all of the function branches and the main branch evolve simultaneously towards a solution to the problem.

One of the problems arising from the use of an ADF-based system is defining the precise shape of the architecture. Specifically, one must decide in advance how many ADFs are present and how many formal parameters each ADF should possess. There are no hard and fast rules, but a precept advocated by Koza is to provide one function for each of the arities 2 to n-1, where n is the size of the terminal set for the problem. For example, a problem with four items in its terminal set would have one ADF with 2 parameters and another with 3 parameters (although there is no requirement that all formal parameters be accessed within the body of a function). Once the ADF architecture has been decided upon, there are no further constraints on the behaviour that may evolve within each ADF, how their arguments are used, and how (or even if) the ADFs are called by the main branch.

The alternative architecture we propose is also hierarchical in nature, but it assigns a more definite purpose to each of the lower-level subsystems. Conceptually, it is quite simple, as shown in Figure 2. Like an ADF tree, this structure also contains a pre-defined number of branches off the root node. Unlike its ADF counterpart, however, there is no 'main' branch. Instead, each branch is charged with the responsibility of handling a *subset* of the input test cases to be applied to the individual as a whole.

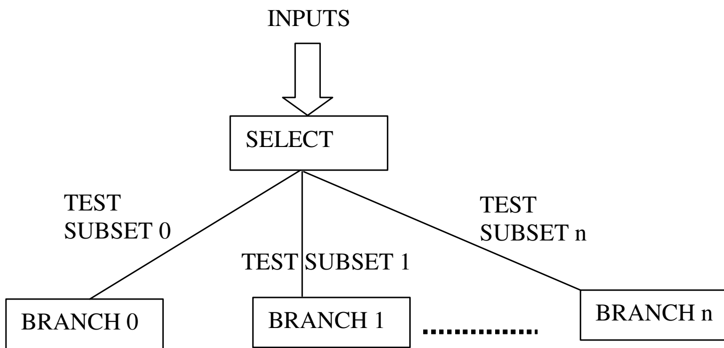


Fig. 2. Overview of the selection architecture

The idea is that, given a set or range of input cases, we partition it into a number of subsets. Code for handling each subset is then evolved independently in separate branches of the program. Decomposition of the original problem in this way should lead to a number of sub-objectives which, in isolation, are easier to solve via evolutionary

computation; a trade-off is an increase in the number of code fragments that must be evolved to solve all branches.

The branches in the selection architecture are not functions in the ADF sense: they are simply code fragments composed from the normal terminal and function sets of the problem. Despite this, each branch does not interact directly with other branches; rather, it is evolved separately and independently. This is a key difference from the ADF architecture, in which all branches evolve simultaneously towards the solution of a problem and the evolutionary value of each branch is judged according to the contribution it makes to the fitness of the individual as a whole. In the selection architecture each branch has its own evolutionary target, its fitness being calculated according to how well it deals with its assigned subset of test cases. Evolutionary effort is focused on one branch at a time rather than all branches at once, although the independent nature of the code fragments means that all branches could readily be evolved in parallel on a multiprocessor machine.

A further difference between the two architectures is that, whereas the number of branches in an ADF system is rather arbitrary, the branch count in the selection architecture is determined by the number of subsets into which the test cases have been divided. These branches are linked together at a single root node, the purpose of which is to decide which branch to activate for a particular combination of inputs. The root node therefore acts as a kind of switch, its exact form depending on the problem being solved and the language being used to encode evolved programs. In most situations it will correspond to a straightforward ‘case’ statement or a nested ‘if-then-else’ construct. The following, for example, is an evolved 4-branch solution to the even-4 parity problem (see next section):

```
SWITCH (INT(D3..D0))
CASE 0..3:
NAND(OR(D0 D1) NAND(D0 D1))
CASE 4..7:
NOR(NOR(OR(D1 D2) OR(D1 D1)) NAND(OR(D0 D1) NAND(D0 D1)))
CASE 8..11:
AND(AND(OR(NAND(OR(D1 D0) AND(D1 D1)) OR(NAND(D1 D0) NAND(D3 D1)))
OR(NOR(NOR(D2 D0) NAND(D2 D2)) OR(D1 D0))) OR(NAND(NOR(D2 NOR(D0 D0))
NOR(NAND(D0 D0) NOR(D0 D2))) NAND(NAND(AND(D1 D2) NOR(D0 D3)) AND(AND(D2
D3) NAND(D3 D1)))))
CASE 12..15:
OR(AND(AND(NAND(AND(D1 D0) OR(D2 D1)) AND(NAND(D0 D3) NAND(D1 D2)))
AND(OR(AND(D2 D1) NAND(D3 D0)) OR(NAND(D1 D3) OR(D2 D3)))) AND(D1 D0))
```

3 Performance

In evaluating the selection architecture, we begin with the even-parity problem, one of a class of Boolean problems that is known to be difficult for GP to solve. In the even-4 version, the aim is to evolve a Boolean design that returns a TRUE output if the number of logic one values on its 4 inputs D0-D3 is even, FALSE otherwise. The parameters for the problem as we have implemented it in our GP systems are given in Table 1.

Table 1. GP parameters for the even-4 parity problem

Objective	To evolve a program capable of determining if the number of logic 1s on the 4 inputs is even
Terminal set	D0, D1, D2, D3
Function set	AND, OR, NAND, NOR
Initial population	Ramped half-and-half
Evolutionary process	Steady-state; 5-candidate tournament selection
Fitness cases	16, representing all combinations of inputs
Fitness	Number of mismatches with expected outputs (0-16)
Success predicate	Zero fitness (solution found)
Other parameters	Pop size=500; Gens=51; prob. crossover=0.9; no mutation; prob. internal node used as crossover point=0.9

The performance of our selection-based system can be compared against a conventional GP system, and also against one which makes use of ADFs. In doing this, we need to make a decision as to how many branches are required, and this in turn is governed by how we choose to partition the test cases. For even-4 parity, exhaustive testing using all combinations of the four inputs {D0, D1, D2, D3} requires 16 test cases. In this experiment, we will assess the effectiveness of having 2 branches, the first dealing with the integer values 0-7 on the four binary inputs, the other dealing with the values 8-15. We will also evaluate the effect of using four branches, dealing with values 0-3, 4-7, 8-11 and 12-15, respectively.

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 2 presents these figures for each of the systems we have described.

Table 2. Performance comparisons for the even-4 parity problem

Approach	**Success rate (%)**	**Comp. Effort**
Standard GP	14	700,000
ADF GP	43	97,500
2 branches, 8 cases each	71	59,500
4 branches, 4 cases each	100	3,500

It is patent that the selection architecture leads to substantially improved performance. When four branches are used, the branches are trivially easy to evolve, several often appearing together in the initial population.

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 given in Table 1 are an additional input D4, a corresponding increase in the fitness cases to 32,

and an increase in population size from 500 to 2000. As before, we experimented with two forms of the selection architecture: one with 4 branches dealing with 8 of the 32 test cases each, and one with 8 branches handling 4 cases each. The results are compared in Table 3.

Table 3. Performance comparisons for the even-5 parity problem

Approach	Success rate (%)	Comp. Effort
Standard GP	0	-
ADF GP	32	864,000
4 branches, 8 cases each	91	192,000
8 branches, 4 cases each	100	16,000

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%. When we try the selection architecture using 4 branches, the success rate is almost triple that achieved in the ADF system, leading to a huge decrease in the computational effort. As before, the use of 8 branches gives us a solution on every run, and an associated computational effort that is comparatively tiny.

To evaluate the approach further, we applied it to the majority-on problem. In this, 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}, but other parameters for the problem as we have implemented it are the same as given for the even-parity problem. For the selection-based approach, we have experimented with 2 branches of 16 cases each, and 4 branches of 8 cases each. The results are given in Table 4.

Table 4. Performance comparisons for the majority-5-on problem

Approach	Success rate (%)	Comp. Effort
Standard GP	62	49,000
ADF GP	7	945,000
2 branches, 16 cases each	90	19,500
4 branches, 8 cases each	100	6,500

An unusual characteristic of this problem is that the performance of the ADF version is substantially worse than that of the conventional GP approach. The superiority of the selection architecture over both of these approaches is evident.

In general, as the number of test cases per branch is lowered, each branch becomes correspondingly easier to evolve, and solutions can be found with a comparatively small population size. At the same time, small test subsets imply a large branch count, so that even though only a small number of generations may be needed to evolve each branch, the maximum number of generations per run may have to be greatly increased

in order to allow enough evolutionary time to generate all branches. Utilising this knowledge allows us to solve comparatively difficult problems by using only small GP populations, simply by extending run lengths to include a sufficient number of generations. A further demonstration of the efficacy of the selection architecture approach is presented in Table 5, which gives the performance figures obtained from solving the even-10 parity problem using a population size of only 2000. To achieve this, the program architecture has been given 256 branches, each handling just 4 test cases, and the maximum run length is set at 500 generations. It is perhaps worth remarking that the figure of computational effort for this approach to the even-10 parity problem is lower than that required for standard GP to solve the even-4 parity problem.

Table 5. Performance of the selection approach on the even-10 parity problem

Approach	Success rate (%)	Comp. Effort
256 branches, 4 cases each	100	628,000

A very different type of problem is that of symbolic regression, in which the GP system attempts to evolve a formula establishing the relationship between two sets of numeric values. Table 6 shows the parameters used for this problem.

Table 6. GP parameters for the polynomial symbolic regression problem

Objective	Symbolic regression of the polynomial $4x^4 – 3x^3 + 2x^2 - x$
Terminal set	x
Function set	+, -, *, /
Initial population	Ramped half-and-half
Evolutionary process	Steady-state; 5-candidate tournament selection
Fitness cases	32 x-values in the range [0, 1), from 0.0 increasing in steps of 1/32, plus corresponding y values
Fitness	Sum of absolute errors in calculated y values
Success predicate	32 hits, a hit being error less than 0.01
Other parameters	Pop size=500; Gens=51; prob. crossover=0.9; no mutation; prob. internal node used as crossover point=0.9

In making use of our selection architecture for this problem we divide the range of x values into equal parts. If, for example, we use two partitions, then one partition handles x values in the range [0,0.5), and the other handles x values in the range

[0.5,1.0). As before, we can make comparisons of performance based on success rate and computational effort. Table 7 shows how conventional GP fares against the selection architecture using 2, 4 and 8 subsets. An ADF system has not been used in these experiments; the reason is, as Koza has reported [2], that while the use of ADFs can be beneficial for sextic polynomials and above, it offers no improvements for quintic and lower order polynomials.

Table 7. Performance comparisons for symbolic regression problem

Approach	Success rate (%)	Comp. Effort
Standard GP	6	1,299,500
2 branches, 16 x-values each	12	1,050,000
4 branches, 8 x-values each	24	535,500
8 branches, 4 x-values each	10	1,876,500

The improvements obtained by the selection architecture are nowhere near as dramatic as they were for the Boolean problems, but are still significant. The 2-branch version does not result in a huge drop in the computational effort needed for standard GP, but does double its success rate. This rate is re-doubled by the 4-branch version, with a halving of the computational effort.

The computational effort statistic provides a rough guide to the amount of work needed to obtain solutions, based on the success frequency and the points during the evolutionary process at which solutions are found. However, it has received criticism regarding its accuracy as an effort prediction tool [16]. For example, although it gives a figure in terms of the number of individuals, it does not take into account differences in the nature of individuals produced by two evolutionary mechanisms, and nor does it consider overheads and other differences in those mechanisms. For this reason, we have gathered further statistics relating to execution runs of the symbolic regression systems.

One way to measure and compare execution performance is to maintain a count of the number of tree nodes evaluated by the fitness function in each of a set of runs. Table 8 summarises the results obtained over a sequence of 50 runs, together with the time elapsed in executing those 50 runs.

Table 8. Comparison of execution statistics for symbolic regression problem

	Standard GP	2-branch	4-branch	8-branch
Avg. nodes/run (x10^6)	558	205	147	83
Max nodes/run (x10^6)	3494	1432	889	355
Min nodes/run (x10^6)	18	9	3	2
Significant?	-	yes	yes	yes
Time for 50 runs (secs)	762	334	233	142

It will be seen that the number of nodes evaluated in the selection architecture is substantially less than the number evaluated in the conventional GP system, and that this number decreases as the number of branches increases. As a result, the elapsed time to execute 50 runs of the selection architecture is also much lower (80% lower in the case of the 8-branch version). This is so even though the selection architecture exhibits better success at finding solutions. In the last-but-one row of the table, the node counts for all 50 runs of the standard GP system are compared against those for the selection architecture using a statistical t-test to validate the significance of the results at the 99% confidence level.

The nature of symbolic regression problems is such that the input data under consideration do not always correspond precisely to a simple polynomial or other formula. Often, one is searching for a formula which provides the closest fit to the curve plotted by the given data. Success rates and computational effort metrics are based purely on evolved programs that satisfy all data points (within the predefined error value). Such programs are all well and good, but what about other, less successful runs? How well do these perform in providing a 'closeness of fit' to the ideal?

The figures in Table 9 are based on the performances of the 'best' programs obtained at the end of each run of the symbolic regression problem. In the problem as we have stated it there are 32 data pairs to be satisfied, and so a correct solution will achieve 32 hits. In runs where this is not achieved, the 'best' program is defined as the individual which attains the greatest number of hits.

Table 9. Comparison of hits attained by best programs in each run

	Standard GP	2-branch	4-branch	8-branch
Avg. best hits	18.8	24.7	24.96	27.28
Max. best hits	32	32	32	32
Min. best hits	6	15	15	22
Significant?	-	yes	yes	yes

The value of 32 in each of the entries for the row headed 'Max best hits' simply indicates that at least one fully correct solution was evolved by each of the GP systems under comparison. More interesting are the other figures, which show that, in general, the selection architecture is capable of evolving programs that are closer to the ideal. This conclusion is also supported by a statistical t-test on the best-hit values obtained across all runs. Particularly interesting is the column for the 8-branch selection architecture: despite having a lower success rate and a higher computational effort value than the other forms of the same architecture, it has a higher average number of hits produced by its best programs. Moreover, every run evolved an individual capable of scoring at least 22 out of the 32 hits available (compare this with standard GP, in which at least one individual reached only 6 hits).

A more visual indication of comparative performance can be obtained by using a scatter graph. Figure 3 plots the best hits obtained for each run, both for standard GP

and for the 4-branch version of the selection architecture. It can easily be appreciated that, in general, the points corresponding to the selection architecture lie significantly above those for standard GP.

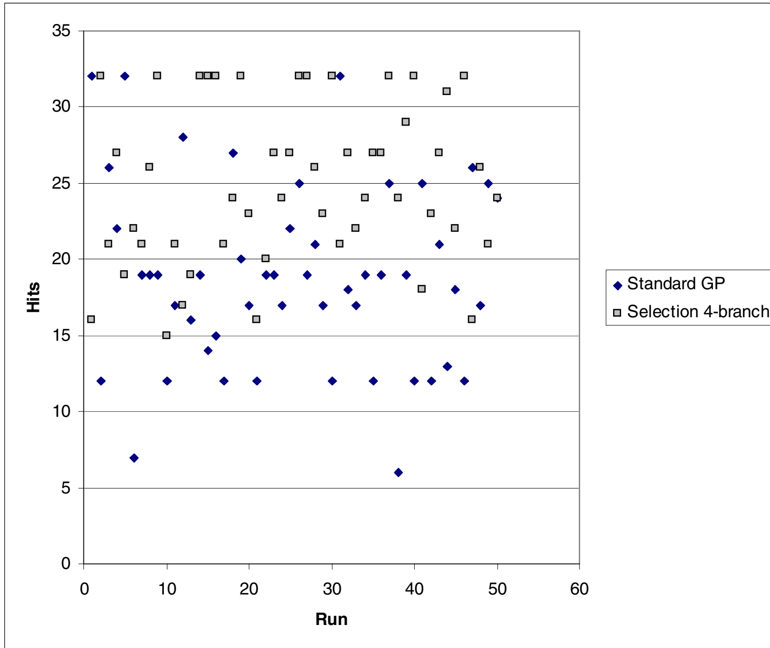


Fig. 3. Scatter plot of hit values attained by best programs in each run

Another characteristic of symbolic regression problems is that the fitness values used to drive evolution do not correspond directly to the hits values. Rather, the fitness of an individual is calculated as the sum of the absolute errors in the calculated y-values. Hence, at any point during evolution, the individual with the best fitness is not necessarily the same as the 'best' individual judged on hit rate. In Table 10, therefore, we also consider the best fitnesses obtained in each of our runs. It should be borne in mind that, with regard to fitness, smaller equals better. As before, a scatter graph (Figure 4) helps to confirm visually the superior performance of the selection architecture.

Table 10. Comparison of best fitnesses achieved in each run

	Standard GP	2-branch	4-branch	8-branch
Avg. best fitness	0.6	0.3	0.25	0.24
Max. best fitness	3.25	0.97	0.76	0.58
Min. best fitness	0.09	0.06	0.09	0.1
Significant?	-	yes	yes	yes

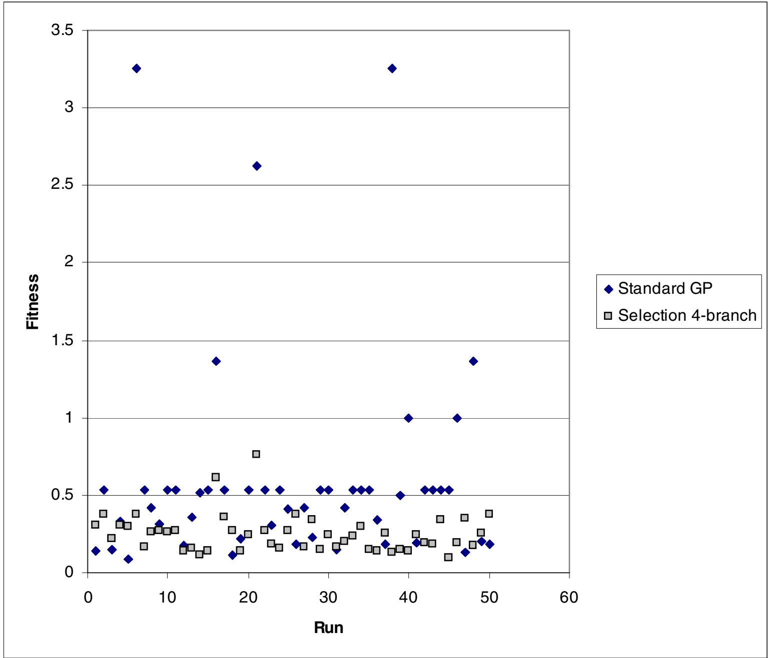


Fig. 4. Scatter plot of best fitness values achieved in each run

4 Conclusions

The selection architecture described in this paper offers significant performance advantages over conventional genetic programming systems. Not only does it lead to a greater success rate at finding solutions, and with less computational effort, it also involves the evaluation of far fewer program nodes, giving a much reduced elapsed execution time. Even in runs where solutions are not obtained, the general fitness of programs that are evolved is much better.

There are other advantages to the use of the selection architecture. The fact that program branches are independently evolved means that branches obtained from differing runs can be combined to form new individuals. For example, the shortest code fragments obtained for each test subset across a sequence of runs can be put together to create a more parsimonious individual. The independent nature of branches also makes it trivial to evolve all branches in parallel on a multiprocessor machine, rather than sequentially as we have done here.

That said, the nature of programs evolved using the selection architecture is very different from that produced by more conventional approaches, consisting as they do of a set of connected relationships rather than a single one across the whole input domain. This gives rise to further intriguing questions regarding the applicability and generality of the approach which we aim to address in future research.

References

1. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
2. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge (1994)
3. Koza, J.R.: Simultaneous Discovery of Reusable Detectors and Subroutines Using Genetic Programming. In: Proc. 5th International Conf. Genetic Algorithms (ICGA-1993), pp. 295–302 (1993)
4. Angeline, P.J., Pollack, J.: Evolutionary Module Acquisition. In: Proc. 2nd Annual Conf. on Evolutionary Programming, La Jolla, CA, pp. 154–163 (1993)
5. Angeline, P.J., Pollack, J.: Coevolving High-Level Representations. In: Langton, C.G. (ed.) Artificial Life III, pp. 55–71. Addison-Wesley, Reading (1994)
6. Rosca, J.P., Ballard, D.H.: Discovery of Subroutines in Genetic Programming. In: Angeline, P., Kinnear Jr., K.E. (eds.) Advances in Genetic Programming 2, ch. 9, pp. 177–202. MIT Press, Cambridge (1996)
7. Roberts, S.C., Howard, D., Koza, J.R.: Evolving Modules in Genetic Programming by Subtree Encapsulation. In: Miller, J., Tomassini, M., Lanzi, P.L., Ryan, C., Tetamanzi, A.G.B., Langdon, W.B. (eds.) EuroGP 2001. LNCS, vol. 2038, pp. 160–175. Springer, Heidelberg (2001)
8. Miller, J.F., Thomson, P.: A Developmental Method for Growing Graphs and Circuits. In: Proc. 5th International Conf. on Evolvable Systems, Trondheim, Norway, pp. 93–104 (2003)
9. Walker, J.A., Miller, J.F.: Evolution and Acquisition of Modules in Cartesian Genetic Programming. In: Keijzer, M., O'Reilly, U.-M., Lucas, S.M., Costa, E., Soule, T. (eds.) EuroGP 2004. LNCS, vol. 3003, pp. 187–197. Springer, Heidelberg (2004)
10. Walker, J.A., Miller, J.F.: Improving the Performance of Module Acquisition in Cartesian Genetic Programming. In: Beyer, H.-G., O'Reilly, U.-M. (eds.) Proc. GECCO 2005, pp. 1649–1656. ACM Press, New York (2005)
11. 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)
12. Gustafon, S.M., Hsu, W.H.: Layered Learning in Genetic Programming for a Cooperative Robot Soccer Problem. In: Miller, J., Tomassini, M., Lanzi, P.L., Ryan, C., Tetamanzi, A.G.B., Langdon, W.B. (eds.) EuroGP 2001. LNCS, vol. 2038, pp. 291–301. Springer, Heidelberg (2001)
13. Hsu, W.H., Gustafon, S.M.: Genetic Programming and Multi-Agent Layered Learning by Reinforcements. In: Proc. GECCO 2002, New York, NY, USA, pp. 764–771 (2002)
14. Hsu, W.H., Harmon, S.J., Rodriguez, E., Zhong, C.: Empirical Comparison of Incremental Reuse Strategies in Genetic Programming for Keep-Away Soccer. In: Deb, K., et al. (eds.) GECCO 2004. LNCS, vol. 3102, Springer, Heidelberg (2004)
15. Jackson, D., Gibbons, A.: Layered Learning for Boolean GP Problems. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 148–159. Springer, Heidelberg (2007)
16. Christensen, S., Oppacher, F.: An Analysis of Koza's Computational Effort Statistic for Genetic Programming. In: Foster, J.A., Lutton, E., Miller, J., Ryan, C., Tettamanzi, A.G.B. (eds.) EuroGP 2002. LNCS, vol. 2278, pp. 182–191. Springer, Heidelberg (2002)

A Comparison of Cartesian Genetic Programming and Linear Genetic Programming

Garnett Wilson[1,2] and Wolfgang Banzhaf[1]

[1] Memorial Univeristy of Newfoundland, St. John's, NL, Canada
[2] Verafin, Inc., St. John's, NL, Canada
{gwilson,banzhaf}@cs.mun.ca

Abstract. Two prominent genetic programming approaches are the graph-based Cartesian Genetic Programming (CGP) and Linear Genetic Programming (LGP). Recently, a formal algorithm for constructing a directed acyclic graph (DAG) from a classical LGP instruction sequence has been established. Given graph-based LGP and traditional CGP, this paper investigates the similarities and differences between the two implementations, and establishes that the significant difference between them is each algorithm's means of restricting inter-connectivity of nodes. The work then goes on to compare the performance of two representations each (with varied connectivity) of LGP and CGP to a directed cyclic graph (DCG) GP with no connectivity restrictions on a medical classification and regression benchmark.

Keywords: Linear Genetic Programming, Cartesian Genetic Programming.

1 Introduction

Genetic programming implementations have been proposed that evolve populations of individuals that are constructed as graphs. Two prominent options in the literature that model GP individuals in this way are Cartesian Genetic Programming (CGP) [1-3] and Linear Genetic Programming (LGP) formulated as a graph structure. LGP in graph form was first presented in [4, 5], with the algorithm for the conversion of imperative instructions to graph formally stated in [6]. The goal of this work was to definitively determine the differences and similarities between CGP and LGP. The comparison motivated an obvious new representation to compare connectivity of the implementations: a directed, cyclic graph (DCG) version of CGP, which is subsequently empirically compared to the original LGP and CGP representations on two types of benchmark problems. The DCG alternative in this paper is simply referred to as "DCG" and is the CGP implementation with the input nodes allowing cycles in the graph. That is, there is simply no restriction on the permitted input nodes: the inputs for a given node may refer to other nodes that occur further "ahead" in the graph, or permit the node to reference itself. Many other more or less elaborate DCG implementations have been introduced in the past, often with the aim of relaxing the restriction of using only feed-forward connectivity to adapt the graphs to real world

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 182–193, 2008.
© Springer-Verlag Berlin Heidelberg 2008

applications. The aim of this paper, rather than to survey graph-based GP approaches, is to investigate the fundamental difference between traditional forms of LGP and CGP, and their restrictions in connectivity.

The following section describes the Cartesian Genetic Programming (CGP) implementation and the components of its representation, with Section 3 describing the implementation and representation of Linear Genetic Programming (in its graph form) in a similar vein. Section 4 compares CGP and LGP implementations to determine their fundamental differences and similarities. Section 5 applies CGP and LGP, each with two parameterizations with differing connectivity constraints, and an unrestricted connectivity DCG, to a classification and regression benchmark.

2 Cartesian Genetic Programming

Cartesian genetic programming (CGP) represents phenotypes of individuals as a grid of nodes addressable in a Cartesian coordinate system. Formally, a Cartesian program is defined by Miller in [3] as the set $\{G, n_i, n_o, n_n, F, n_f, n_r, n_c, l\}$ where G is the genotype that is a set of integers to be described, n_i is the indexed program inputs, n_n is the node input connections for each node, and n_o is program output connections. The set F represents the n_f functions of the nodes, and n_r, n_c are the number of nodes in a row and column, respectively. The levels back parameter, l, indicates how many previous columns of cells have their outputs connected to a node in the current column (with primary inputs treated as node outputs). Program inputs are permitted to connect to any node input, but nodes in the same column are not allowed to be connected to each other. Any node can be either connected or disconnected. See Figure 1 for a diagram of a typical CGP graph.

A graph of the individual consists of a string of integers specifying, firstly, n_n inputs and one internal function for each node, and lastly the n_o program outputs. The CGP genotype thus takes the form of the string of integers

$$C_0, f_0;\ C_1, f_1; \ldots ; C_{cr-1}, f_{cr-1}; O_1,\ O_2,\ \ldots, O_m \tag{1}$$

where C_i indicates the points to which the inputs of the node are connected, and each node is given an associated user-defined function f_i. It is possible to have a list composed of functions with different arities by setting the node arity to be the maximum arity present in the function list and allowing nodes that require fewer inputs to simply ignore the extra inputs. Node 0, described by C_0, f_0, always has an output label that is one greater than the number of program inputs (denoted n in Figure 1). There are also m output genes O_i corresponding to the m program outputs.

In principle, CGP is capable of representing directed multigraphs but has only been used thus far to represent directed acyclic graphs (DAGs). If CGP only encodes DAGs, then the set of possible alleles for C_i are restricted so nodes can only have their inputs connected to either program nodes from a previous (left) column or program inputs. (In other words, they have “feed-forward” connectivity.) As stated by Miller

and Smith [2], in many actual CGP implementations the number of rows (r) is set to one, and thus the number of columns (c) is the maximum allowed number of nodes. The levels-back parameter (l) can thus be chosen to be any integer from one to the number of nodes in the graph (n). The output genes are also unnecessary if the program outputs are taken from the m rightmost consecutive nodes when only one row is used. The generic form of CGP is presented in Figure 1 (left), along with typical practical restrictions (right).

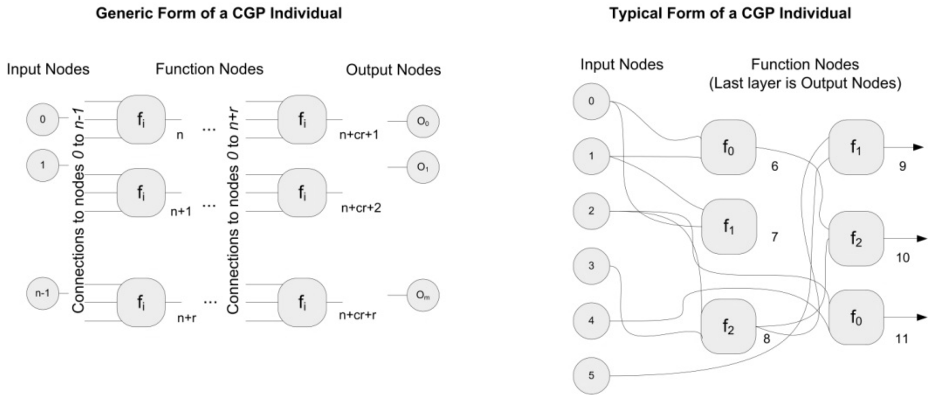


Fig. 1. The generic (left) and typical (right) CGP representations where f_i is a member of the function set, n is the number of inputs, m is the number of outputs, c is the number of columns, and r is the number of rows

To relate CGP in practice to CGP as originally defined in [3], we recall that a CGP program is formally defined by

$$\{G, n_i, n_o, n_n, F, n_f, n_r, n_c, l\} \tag{2}$$

For simplicity, since we are interested in the final graphical representation of a CGP individual, we can eliminate G (the integer representation of the graphical elements n_o, n_n which is redundant for the purposes of representing only the components of a CGP graph) and F (the set of user-defined functions that will be represented as nodes) that are not themselves components of the graph. The CGP graph is now represented

$$\{ n_i, n_o, n_n, n_f, n_r, n_c, l\} \tag{3}$$

In practice, the n_o program outputs need not be used in a graph representation [2]. Instead, recall that some $m < c$ rightmost consecutive nodes for c columns provide outputs when one row is used. This eliminates the need of the n_o variable in the graph representation. Also, when the number of rows is set to one, the number of nodes in a

column n_c will always be one and it is necessarily the case that $n_r = c$ (where c is the number of columns = number of internal nodes). Finally, the levels back parameter is often set to be the number of columns, c (as in [2]) to allow a given node to connect to any previous node, but l can be set to any integer k, $k < c$. Making appropriate substitutions, this gives us the typical graph representation of

$$\{ n_i, 0, n_n, n_f, 1, c, l \} \tag{4}$$

3 Graph Representation of Linear Genetic Programming (LGP)

In linear genetic programming (LGP), the genotype individuals have the form of a linear list of instructions as a binary string [6]. This binary string may in turn be interpreted or represented as a set of integers, just as in CGP genotype representations. Program execution is that of a simple register machine (Von Neumann computer), and instructions are made up of opcodes and operands (providing linear forms of Functional and Terminal sets, respectively). As the program executes, it alters the contents of the internal registers (or stack) and solution register(s).

When the bit strings are interpreted, they correspond to members of the Functional (and sometimes Terminal) sets to produce a phenotype solution. For instance, the binary sequence "011" in the individual's genotype could be interpreted as the functional set member "addition" in the phenotype. The immediately following bits often refer to destination and source registers, if registers are used as opposed to a stack. The phenotype is then evaluated to determine the corresponding fitness. The structure of a linear GP individual is depicted below in Figure 3.

The instruction sequence (imperative) view of a linear program can be transformed into an equivalent functional representation in the form of a directed acyclic graph (DAG). This is simply an alternate way of representing the linear program and registers. The directed nature of the graph better enables the deciphering of functional dependencies and execution flow during interpretation of the instructions. For details of the formal algorithm to convert LGP to a DAG, the reader is referred to [6]. The application of the algorithm to imperative instructions produces a DAG such that the number of inner nodes always equals the number of imperative instructions. Each of these inner nodes includes an operator, and has as many incoming edges as there are operands for that operator in the corresponding imperative instruction. Sink nodes have no outgoing edges and are labeled as registers or constants. While the nodes in [6] are labeled with only operators, the nodes are plotted as unique nodes in virtue of target register and operator at a particular execution point. The maximum number of sink nodes is thus the total number of registers and constants in the terminal set. Upon completion of the DAG, the sinks represent input variables of the program. Constant sinks and inputs may be pointed to from every program position. An LGP program in the form of binary genotype, interpreted program, and graph structure is shown in Figure 2 below.

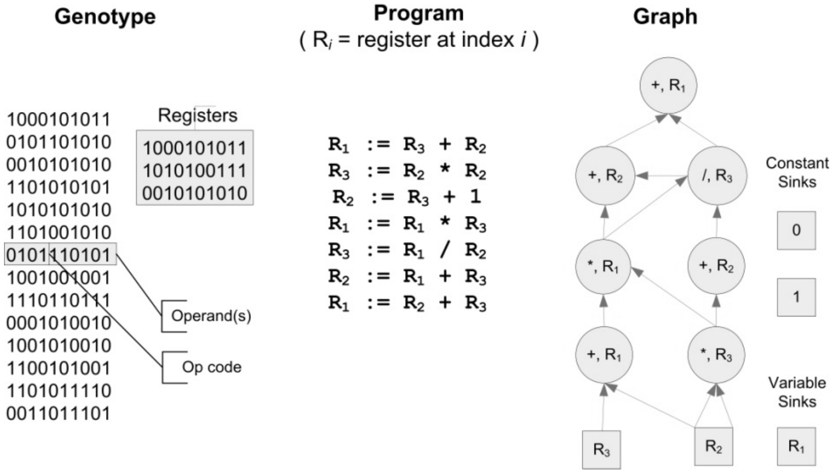


Fig. 2. Representations of a LGP individual

4 Comparison of CGP and LGP Representations

We must now determine whether or not LGP and CGP graphs are similar representations, and if they are, what is the nature of that similarity? In the case of both graphs, each unique node is identified by a function and the nodes from which input to the function is received. Again, in both cases, inputs for a given inner node can only be received from nodes or variable/constant sinks (inputs for CGP) that have already been established in the graph. The semantic representation of the nodes is thus relevantly similar between the two representations.

Using the seven-tuple $\{n_i, n_o, n_n, n_f, n_r, n_c, l\}$ representation of CGP, how would LGP be formulated? In LGP, the constant and variable sinks are effectively program inputs, n_i. By specifying the output to be the content of an LGP register at the finish of imperative instruction execution, n_o is set to 0 as in common CGP practice. That is, there are no special output nodes in either CGP or LGP. There are a given number of internal nodes in LGP specified by input connections and functions; let us specify these as n_n just as in CGP implementations. There is also a function set consisting of n_f functions. As in the most common CGP implementations, there is no notion of separate rows and columns in LGP, so there are $n_c = c$ and $n_r = 1$ nodes in an LGP graph. If nodes are capable of being connected to any previous node in LGP (as is typical in CGP), then $l = c$. However, the usual in LGP (as presented in [6]) is that a given node can only connect to the particular nodes in the previous layers *that last used the registers specified by the function in the current node.*

Given the considerations thus far in this section, the tuple representing a LGP graph contains the same representative elements as typical CGP (Eq. 4), with the exception that nodes in LGP graphs take input from previous nodes that last used the registers the current node requires as inputs. Thus, the levels-back parameter (l) of the tuple is not relevant to LGP graphs, giving the tuple where l is not applicable (*n/a*):

$$\{\ n_i,\ 0,\ n_n,\ n_f,\ 1,\ c,\ \text{n/a}\} \quad (5)$$

The representation elements of typical LGP and typical CGP in practice really only differ in the parameter of the number of levels back. (Compare Eq. 4 and Eq. 5.) In a practical sense, this means that there are different restrictions on how a given inner (function) node in the two implementations can refer to incoming nodes. The interconnectivity of the LGP graph is thus constrained in an implicit way, as opposed explicitly specifying the levels-back parameter in CGP.

In terms of genotype representation, a CGP genotype is a series of pairs representing nodes. Each pair consists of a set of points to which the inputs of the node are connected, and the function for the node. Upon listing the connections and functions for all nodes, the nodes from which the output(s) are to be taken are specified in the genotype (see Section 2), or they may simply be specified as some number of last nodes in the graph as a parameter outside the genotype. This is largely a design decision, but the specification of output nodes may or may not be under control of evolution as part of the genotype in CGP. In contrast, an LGP implement-tation typically chooses particular register(s) in which the output is to be found, and the output registers are not listed in the genome. As mentioned previously, the output nodes are typically taken to be the last layer of graph nodes in CGP, so output nodes are effectively left out of the CGP genotype in modern representations (making the genotype similar to LGP in that the nodes specified for output are not part of the genotype). In the case of both LGP and CGP, one can also have a number of outputs from the registers or nodes, respectively.

An LGP individual's genotype is a list of imperative instructions. Each line represents a function and has some associated registers and a destination register. Using the algorithm of Brameier and Banzhaf, though, the genotype representation can be converted to a graph, which can alternately be described as a listing of nodes including function and input edges. A node is made unique in virtue of three components: source register(s) (or source data) and destination register, function, and when it is executed in the program (placed in the graph). The source register(s) or source data effectively indicate nodes to which the incoming edges are connected because the last nodes having used the source registers of the current node as their target register (or simply the specified input data from variable or sink nodes) will be connected to the incoming edges. The destination register serves to reference the output edge of the node because the next future nodes to reference the current node's destination register as source registers will form an incoming edge from the current node. The final value in the register(s) of interest at the end of execution in LGP are the output value(s) in LGP, and they are the last instances of the nodes labeled with the relevant registers in an LGP graph. Note that this has the same effect as choosing a particular node (or nodes) as the output in CGP, which is what is done typically in current implementations. Also, nodes are actually labeled with only the instruction operator, and the target register can be added to the node label for clarity but is generally used as a temporary variable to plot the LGP graph (see [6]). In this work, the target registers are included on the node labels for clarity of interpretation, but they are generally left out the of the final plot (as in [6]).

Functions are encoded in the same way in both CGP and graph LGP. Collectively, the components of a single instruction in an LGP genotype correspond to a node that is a unique imperative instruction. The only difference in the encoding of the LGP and CGP graph is that there is an explicit identification of the node for future outgoing edges that is encoded in the genome in LGP (in virtue of the instruction's target register), whereas in CGP the nodes are just sequentially ordered as they appear and not encoded as part of the genome. This means that the encoding of the genome restricts what previous nodes get connected to a node in a current layer in graph LGP. In contrast, in CGP that restriction is handled by specifying the levels back (l) parameter, and it is not explicitly coded in the genome. Thus, in LGP the connectivity of the nodes is under evolutionary control since it is part of the genome, but in CGP it is specified *a priori* as a design parameter.

The characteristics of the elements required for graph representation and the genotype structures of CGP and LGP dictate their graphs will be similar (Eq. 4 and Eq. 5). Consider the common CGP using one row and the LGP graph for programs of single non-conditional, non-branching imperative instructions. Typically, these graphs will both involve two inputs per node if Boolean functions such as AND, NAND, OR, and NOR as are typically used in circuit board design are used. However, in both CGP and LGP graphs, the nodes may accept varying numbers of input edges depending on the maximum required by the function with the most arguments in the function set. Furthermore, both graphs are directed, with data only flowing in the direction from input nodes/sinks to output nodes. In other words, programs are restricted such that nodes only have their inputs connected to the program inputs or nodes from a previous column; edges only point in the general direction of the output. To summarize, both CGP and DGP are represented as DAGs with each node capable of any number of input edges. The only difference between CGP and DGP graphs is the restriction on how the input edges are assigned to a node (as discussed in the previous section). Thus, given an unlabelled DAG with arbitrary node layout generated by either GP variant, the user could not readily distinguish between the two without further information, namely the design parameterization of the CGP tuple $\{n_i, n_o, n_n, n_f, n_r, n_c, l\}$ and the number of registers used in the LGP algorithm. See Table 1 below for a summary of the comparison of LGP and CGP graph representations.

Table 1. Comparison of representation components of CGP and LGP (differences in bold)

	CGP Graph	Graph LGP
Tuple	$\{ n_i, 0, n_n, n_f, 1, c, \boldsymbol{l}\}$	$\{ n_i, 0, n_n, n_f, 1, c,$ **n/a**$\}$
Genotype	Integer or binary string	Integer or binary string
Graph Type	DAG	DAG
Node content	Function	Function
Connectivity	**Restricted by levels-back (not under evolutionary control)**	**Restricted by usage of target registers (evolutionary control)**
Incoming edges	Maximum required by function set	Maximum required by function set

5 Comparison of Graph-Based Genetic Programming Techniques on Classification and Regression Benchmarks

In previous sections, the representation elements and their implications for the functionality of the CGP and LGP graph types was discussed. The main difference between LGP and CGP was the mechanism used to restrict the allowed input edges to a given node, including whether or not the edges are under evolutionary control. To provide contrast to, and determine the practical value of, the connectivity restrictions of CGP and LGP, we introduce a new graph type called simply "DCG" for "directed cyclic graph." This new graph type follows the CGP representation, only that each node can accept inputs from any node in the graph. This means that there is no restriction of data flow to only feed-forward connectivity, cycles are permitted, and the levels back parameter is not relevant. LGP graphs can also permit cycles, but the corresponding imperative LGP programs would have to involve jump statements. Such considerations are beyond the scope of this work, as the LGP would no longer conform to the current formal algorithm in [6] which is used here for the comparison of traditional CGP and graph LGP.

Two implementations of CGP with varying connectivity are tried, with levels back being equal to the number of columns (nodes), or only 2. In LGP, two progressively constricting instruction forms are tried: 1 input and 2 input. In the single input implementation, an instruction applies a function to data from a source register X and target register Y, replacing the data in that same target register Y. In the two input implementation, an instruction applies a function to data from two source registers X and Y, placing the result in another target register Z. In addition, the number of inner nodes in LGP graphs is determined by the nature of the instructions: Due to the use of registers in LGP, functions in nodes may draw their input(s) from registers or fitness cases. If there is a larger number of fitness cases than registers (as in the classification benchmark), fewer bits are needed to specify one of four registers, but more bits are needed to load from one of the fitness case fields. In the regression benchmark, there are fewer fitness case features than number of registers. Whether or not to load from register or fitness case is determined by a binary flag. Only the required number of bits is used to interpret a given instruction, resulting in individual-dependent graph sizes. The summary of the general parameterization of the implementations is given in Table 2.

Table 2. General parameterization of CGP, LGP, and DCG implementations

Tournament Style	Steady State, 4 individuals per round
Population size	25
Genotype structure	240 bit string, 4 registers (LGP)
Graph structure	16 inner nodes + input nodes (CGP & DGP), determined by bit string (LGP)
Genotype mutation	point mutation, threshold = 0.9

Here we compare the graph GPs' empirical performance on a real world classification benchmark, namely the Heart Disease data that is part of the UCI Machine Learning Repository [7], and the Mexican Hat regression benchmark as

described in [6]. Only mutation is used in these experiments, as crossover in CGP does not make intuitive sense. Furthermore, mutation is appropriately restricted in CGP so that a given node can only refer to previous nodes in the graph according to the levels back parameter. Naturally, mutation is unrestricted in LGP and DGP. Execution is carried out from inner start node to end node in CGP and DCG (execution sequence is already determined by order of instructions in LGP). To allow data (other than default) placed in a node to be fed back through the network in DCG, multiple execution iterations over the inner nodes is required. In DCG experiments, five iterations of the inner nodes are executed per fitness case.

The medical database contains 303 instances (164 negative, 139 positive), each consisting of 13 attributes, with a 14th indicating positive or negative diagnosis. Prior to trials, unknown values were replaced by the mean value of the relevant attribute and the positive or negative diagnosis was changed to '1' or '0', respectively. The results use four-fold cross-validation to verify accuracy of the findings. Each partition consisted of a unique 25% test set and 75% training set and retained the class distribution of the entire data set. If the output of an individual was less than 0 on a fitness case, the case was classified as a negative diagnosis; otherwise the individual classified the case as positive. The function set used was { +, *, -, /, SIN, COS, EXP, NATLOG }, protected as appropriate. Fitness was defined simply as number of correct classifications, and training was conducted for 30 000 rounds. The results are shown in Figure 2. The median and spread shown in the boxplot correspond to the mean accuracy across the four unique test sets used in four-fold cross-validation over the 50 trials. Each box indicates the lower quartile, median, and upper quartile values. If the notches of two boxes do not overlap, the medians of the two groups differ at the 0.95 confidence interval. Points represent outliers to whiskers of 1.5 times the interquartile range. A customized version of the popular Java-based Prefuse [8] framework was created a provide a means of visualizing the final graph topologies, where the best trial in each implementation for the first partition are shown in Figure 3.

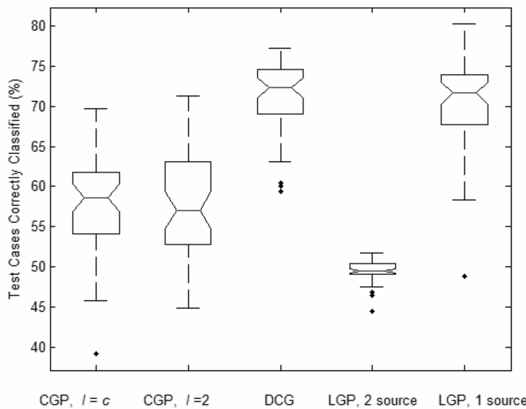


Fig. 3. Boxplot of mean classification accuracy for the Cleveland Heart data set over 50 trials using four-fold cross-validation. Each partition was 75% training, 25% test.

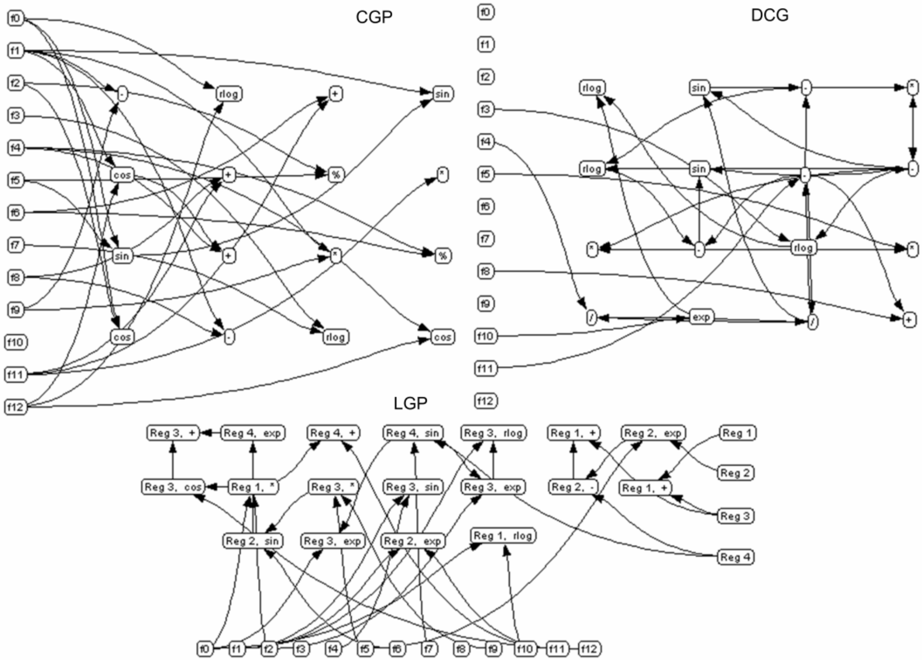


Fig. 4. Individuals corresponding to the best final solution for CGP ($l = c$), LGP (2 input), and DCG types for the best trial in a particular partition of the UCI Machine Learning Repository Heart Disease test set. The node corresponding to final classification is the lower right-hand node in the CGP and DCG graphs, and the upper left node in LGP.

Figure 2 indicates that the chosen DCG model (center) outperformed the more restrictive and traditional Cartesian GP implementations (two leftmost). DCG was not outperformed by the more restrictive form of LGP (rightmost), with no statistically significant difference shown (note overlapping notches.) The least restrictive LGP implementation did not perform as well as the other implementations (second from the right). The additional freedom of data flow within the DCG graph due to the admission of cycles enhanced classification ability. Furthermore, in all cases, the restriction of information flow within implementations of both CGP and LGP models led to decreased classification accuracy. In figure 3, the directed edges in the DCG solution show that it clearly takes advantage of its freedom of connectivity and admission of cycles.

The two- Mexican Hat problem as described in [6] is tested on the implementations to provide a regression benchmark. The problem is named for the shape of the three-dimensional plot of its function

$$f_{mexicanhat(x,y)} = (1 - \frac{x^2}{4} - \frac{y^2}{4}) \times e^{(-\frac{x^2}{8} - \frac{y^2}{8})} \qquad (6)$$

Following the parameterization of [6], 400 fitness cases were used, with the input range restricted to [-4.0, 4.0]. The function set consisted of {+, -, x, /, x^y}, protected

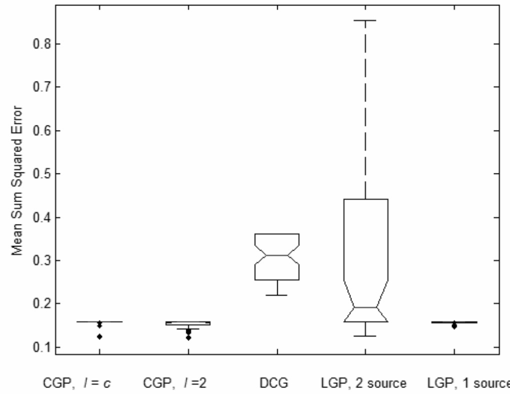


Fig. 5. Boxplot of mean sum squared error for the Mexican Hat problem set over 50 trials

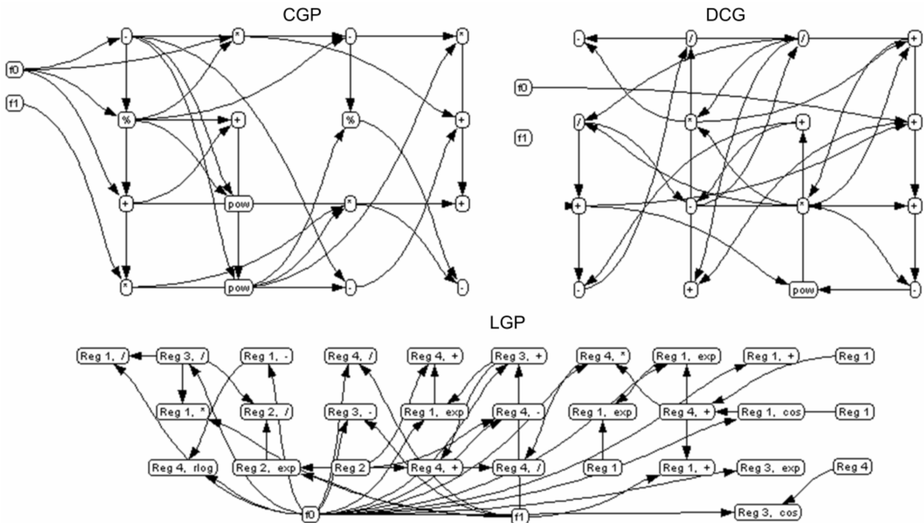


Fig. 6. Individuals corresponding to the best final solution for CGP (l = c), LGP (2 input), and DCG types for the best trial for the Mexican Hat problem. The node corresponding to final classification is the lower right node in CGP and DCG, and the upper left node in LGP.

when needed, and tournaments ran for 1000 rounds. Figure 4 shows the boxplot of the mean sum squared errors of the implementations for the Mexican Hat problem, with best final networks of CGP, DCG, and LGP shown in Figure 5.

Given the regression benchmark (Figure 4), DCG clearly does not perform as well. It is also noteworthy that the unrestricted LGP has the greatest variability over all solutions. Comparing classification and regression benchmark performance (Figures 2 and 4), it is evident that less restricted connectivity (DCG and 2 source LGP) is not of benefit in this regression benchmark. Furthermore, all CGP variants and the

restricted LGP variant perform best on the regression benchmark but worse on the classification. This difference may be due to the freely connected nature of DCG allowing it to provide a more highly adapted configuration for classification of a complex problem. In contrast, the definite answer in the regression problem, to be found within a lower number of tournament rounds, is hindered by the greater availability of configurations and cycles in the DCG. In Figure 5, we see that this is the case where the best DCG incorporates extensive connectivity while processing only one input, whereas CGP does not maximize its available levels-back flexibility.

6 Conclusions and Future Work

This work establishes that the difference between graph-based LGP and CGP is the means with which they restrict the feed-forward connectivity of their DAG graphs. In particular, CGP restricts connectivity based on the levels-back parameter while LGP's connectivity is implicit and is under evolutionary control as a component of the genotype. Unrestricted forms of LGP and CGP, and DCG, performed well on the real world medical classification benchmark, but the flexibility of the less restricted graph types did not allow them to perform as well on a regression benchmark. In future work, we plan to explore GP-based search using DCGs in an industry-based real world application. Possibilities for future investigation also include a DCG analogy of LGP graphs, and a closer examination of the relationship between performance of the representations and their connectivity characteristics and evolvability.

Acknowledgements. The authors gratefully acknowledge the support of a PRECARN Postdoctoral Fellowship.

References

1. Miller, J.F., Job, D., Vassilev, V.K.: Principles in the Evolutionary Design of Digital Circuits - Part 1. Genetic Programming and Evolvable Machines 1, 8–35 (2000)
2. Miller, J.F., Smith, S.L.: Redundancy and Computational Efficiency in Cartesian Genetic Programming. IEEE Transactions on Evolutionary Computation 10, 167–174 (2006)
3. Miller, J.F., Thomson, P.: Cartesian Genetic Programming. In: Poli, R., Banzhaf, W., Langdon, W.B., Miller, J., Nordin, P., Fogarty, T.C. (eds.) EuroGP 2000. LNCS, vol. 1802, pp. 121–132. Springer, Heidelberg (2000)
4. Banzhaf, W., Nordin, P., Keller, R., Francone, F.: Genetic Programming - An Introduction: On the Automatic Evolution of Computer Programs and Its Applications. Morgan Kaufmann Publishers, San Francisco (1998)
5. Nordin, P.: Evolutionary Program Induction of Binary Mchine Code and its Application. Krehl Verlag, Munster (1997)
6. Brameier, M., Banzhaf, W.: Linear Genetic Programming. Springer, New York (2007)
7. Newman, D.J., Hettich, S., Blake, C.L., Merz, C.J.: UCI Repository of Machine Learning Databases. University of California, Department of Information and Computer Science, http://www.ics.uci.edu/~mlearn/MLRepository.html
8. Heer, J.: Prefuse Interactive Information Visualization Toolkit, http://prefuse.org

Evolvability Via Modularity-Induced Mutational Focussing

Richard M. Downing

University of Birmingham
rmd@cs.bham.ac.uk

Abstract. This work postulates a mechanism by which random genotypic variation is directed towards favourable phenotypic variation. Evolvability is a poorly understood concept at present: it is unclear precisely how the genotype-phenotype map aligns random genotypic mutation with favourable phenotypic variation. By static analysis of the distribution of the genotypic representation of functionality, an emergent bias in the representation of the adapted and maladapted is shown. This bias is facilitated by a form of reuse modularity, and it serves to direct phenotypic variation to where there is selective opportunity.

1 Introduction

Organisms in nature generate offspring that are both highly viable and which vary in dimensions for which there is selective opportunity. This phenotypic variation is not random, though it is generated by random genotypic variation. Kirshcner & Gerhart's [10] theory of *facilitated variation* states that it is the organism itself that is central in directing phenotypic variation. However, the mechanisms by which this is achieved are not fully understood.

Systems of artificial evolution lack the capability to direct phenotypic variation in a manner so readily exhibited in nature: they lack evolvability. Altenberg [2] states of evolvability:

> "It comes from the genetic operators being able to transform the representation in ways that leave intact those aspects of the individual that are already adapted, while perturbing those aspects which are not yet highly adapted. Variation should be channeled toward those "dimensions" for which there is selective opportunity."

But how, and what are the properties and mechanisms that facilitate it? For evolvability to be applied to systems of artificial evolution, it must first be understood and characterised.

In a trivial representation in which there is a one-to-one correspondence between genotypic and phenotypic features, mutation is just as likely to affect one feature as it is any other, so variation is not 'channeled' at all. In a non-trivial representation, for which there is not a one-to-one correspondence between genotypic and phenotypic features, the genotypic representation of adapted and maladapted functionality within the genotype overlaps. Under that assumption, the

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 194–205, 2008.
© Springer-Verlag Berlin Heidelberg 2008

variation operators must, it would seem, have some mystical property to change the representation in a way that perturbs the maladapted but leaves intact the adapted: understanding evolvability can therefore appear insurmountable.

This paper hypothesises a mechanism by which directed phenotypic variation may be facilitated. The hypothesis is referred to as *Evolvability via Modularity-induced Mutational Focussing* (EMMF). EMMF postulates that a biased distribution of adapted and maladapted functionality emerges within the genotype under the forces of random mutation and selection. Such a bias greater exposes the maladapted functionality to perturbation while protecting the adapted. By this, variation is channeled towards dimensions for which there is selective opportunity, fulfilling Altenerg's requirement for facilitating evolvability.

2 Representation and Algorithm

For this work, Downing's approach of *Evolving Binary Decision Diagrams with Inherent Neutrality* (EBDDIN) [7] is employed. Binary Decision Diagrams are first introduced, followed by an overview of the EBDDIN algorithm.

Introduced by Lee [11] and further by Akers [1], a Binary Decision Diagram (BDD) 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 $\mathbb{B}$. Given an assignment of values for V, the output is determined by traversing the BDD from the root to a terminal following the child indicated by each vertices' variable label value.

Bryant [6] introduced the *ordered* BDD (OBDD), which imposes a total ordering on the appearance of non-terminal labels along any path 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.

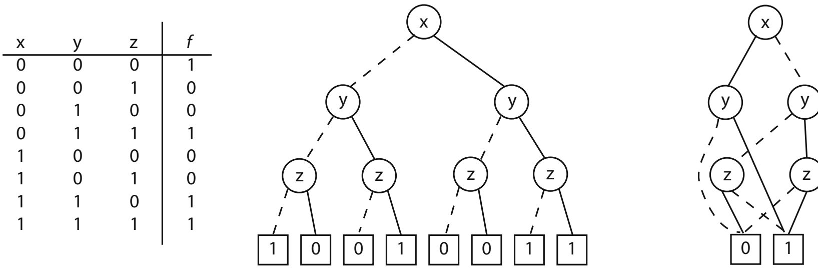
The OBDD representation is derived from the Shannon expansion [12]. A Boolean function $f(x_1, \ldots, x_n)$ is decomposed into subfunctions using a specified ordering of variables, represented by vertices in the OBDD, thus:

$$f = x_i \cdot f|_{x_i=1} + \bar{x}_i \cdot f|_{x_i=0}$$

where $f|_{x_i=b\in\mathbb{B}}$ is the restriction of x to the constant $b \in \mathbb{B}$. The decomposition of the subfunctions continues until the Boolean constants are reached. The resulting list of expressions may contain some redundancy, i.e. duplicate expressions (see [3] for further details).

Redundancy in an OBDD can be removed in two ways:

1. **Remove Redundant Tests.** A nonterminal α that has both outgoing edges pointing to the same vertex β is redundant. Redirect all α's incoming edges to β.



x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Fig. 1. Truth table, BDT, and ROBDD representations of the function $f(x, y, z) = \bar{x}\bar{y}\bar{z} + xy + yz$. The BDT is easily created from the truth table. The BDT is then reduced to the canonical ROBDD form by applying the reduction mechanisms mentioned in the text. Many intermediate OBDDs are created in the reduction process.

2. **Remove Duplicate Vertices.** If α and β are nonterminals and have the same variable label and same children, or α and β are terminals of the same value, one can be removed with its incoming edges redirected to the remaining vertex.

A *reduced* OBDD (ROBDD) is an OBDD that cannot have its complexity reduced further by the reductions described above. Bryant [6] has shown ROBDDs to be *canonical forms*: each function has a unique ROBDD representation for each π, allowing easy equivalence and satisfiability checking. Figure 1 shows three different representations of the same function. The Binary Decision Tree (BDT), which is a specific form of OBDD, is easily constructed from the truth table, as every input vector (i.e. line in the truth table) has its own path in the BDT. The BDT is reduced to the canonical ROBDD representation by repeatedly applying the reduction mechanisms described above; many intermediate OBDDs are generated in the process.

The variable ordering 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 6-bit multiplexer produces an ROBDD having complexity 7 while the the worst π results in and ROBDD having complexity 29 (see figure 2); for the 11-bit multiplexer, the best and worst ROBDD complexities are 15 and 509 respectively; for the 20-bit multiplexer it is 31 and over 130,000 respectively. For the n-bit multiplexer, the complexity grows linearly for the best π and exponentially for the worst. In general, the variable ordering problem is NP-complete in both exact and approximate solutions [5,13].

The EBDDIN approach to evolving BDDs employs four neutral mutations derived from the reduction mechanisms above and their inverses, and also a mutation that redirects a child edge to a new vertices. See figure 3. In this paper only a static variable ordering is considered. The test function employed is the 11-bit multiplexer function, and it is used two configurations: the first having the best ordering and ROBDD complexity 15 (11-mux); the second having the worst

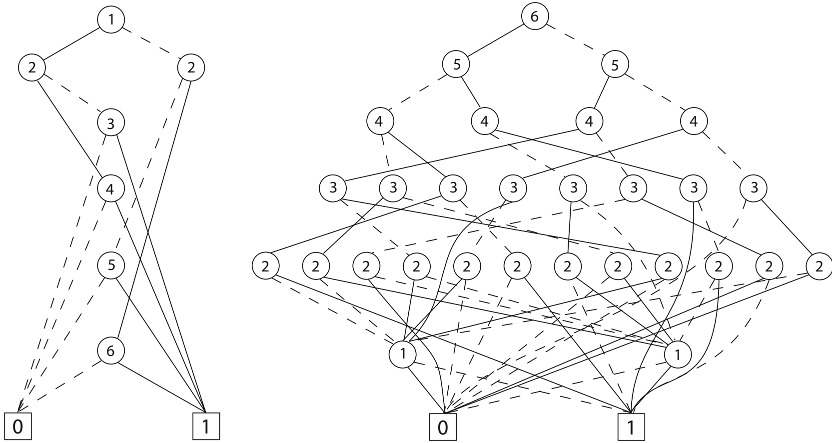


Fig. 2. The effects differing π on ROBDD complexity for the 6-bit multiplexer function. Both of the above ROBDDs are representations of the same function but have different variable orderings. ROBDD complexity for this function is very sensitive to π, having a linear complexity in the number of variables for the best π, but exponential for the worst π.

and reverse ordering and ROBDD complexity 509 (11-mux(R). This function was chosen specifically because of its extreme sensitivity to changes in π, and should better emphasise the property under study. However, any function with similar sensitivity to π would suffice. Further details on EBDDIN can be found in [7].

3 Modularity

Many EAs incorporate modularity to their reported benefit. Modularity can help generalisation, facilitate reuse, and help prevent disruption by variation operators [4]. Furthermore, Woodward [14] has shown that, in the presence of modularity and equal expressitivity, the complexity of solution is independent of the chosen function set.

An OBDD can be thought of as a more general form of a tree in which subtrees are shared and reused in a modular fashion. In this respect, an OBDD is somewhat like a tree in Koza's GP with inherent ADFs. For a function with a compact ROBDD representation there are a plethora of neutral variants, each exhibiting different degrees and configurations of modularity. These neutral variants occupy the space between the ROBDD and BDT representations of the function. Each neutral mutation either removes or adds one vertex to the size of the OBDD, and there will typically be several variants of even a given size. Thus, the neutral evolution of modularity under EBDDIN is a gradual matter under a minimal mutation severity, with many degrees of freedom and a potentially massive number of configurations to explore. The potential for modularity to evolve in EBDDIN is, therefore, significant.

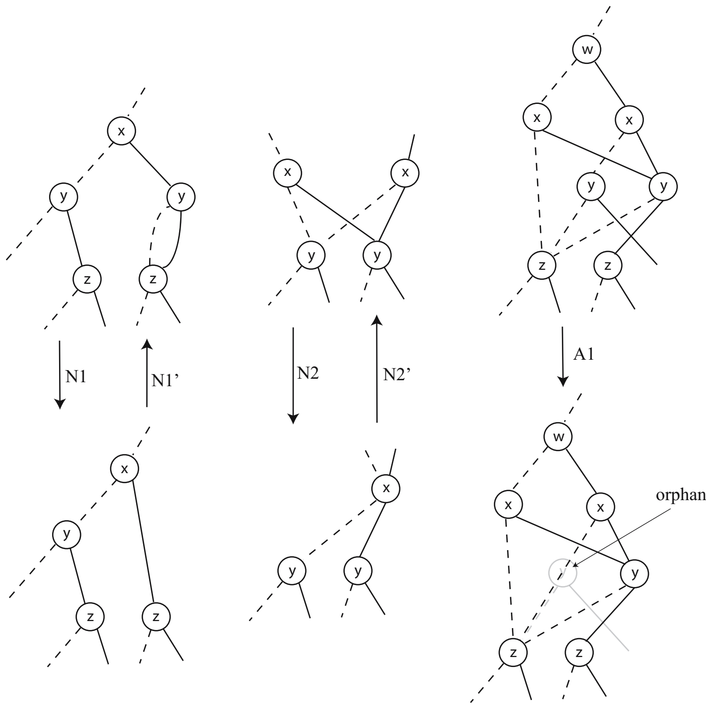


Fig. 3. The five mutations of EBDDIN, four neutral and one functionally modifying. N1/N1': remove or insert redundant test; N2/N2': merge nodes with identical label and children, or split; A1: redirect an edge to another vertex at a lower level to the parent.

What the different configurations represent are different alignments between the spaces of phenotypic variation and fitness. Favourable configurations will better facilitate perturbation of the maladapted functionality while leaving intact the adapted functionality. If such configurations can emerge in evolution then evolvability will be enhanced, and this is what the EMMF hypothesis predicts.

4 Modularity and Punctuated Evolution

Before going onto investigate the EMMF hypothesis in detail, a comparison of fitness curves on both modular and non-modular representations of 11-bit multiplexer is given. Figures 4 and 5 show the fitness curves for the best individual for 11-mux and 11-mux(R) using a (5+10) ES. There is a very clear contrast here. The figures show that it is not only the pace of evolution that differs, but also the manner. While 11-mux(R) exhibits a fairly consistent gradualistic curve, 11-mux has much more erratic and variable behaviour. Long periods of stasis are interspersed with periods of rapid evolution, and this is indicative of Gould's [9] punctuated equilibria phenomenon.

Two of the main stasis points are indicated in figure 4. These points of stasis happen at fitness levels of -128 and -256. Once a stasis point is broken free of, fitness increases at a rate not appearing to significantly decrease from the rate prior to stasis. The implication is that the solution has emerged in component parts consistent with the subfunctions of the ROBDD target representation, or some fraction thereof. The stasis points are indicative of higher-level components the foundations of which have not yet emerged. Once the foundation of the missing functional component is discovered, functional evolution continues apace, exploiting preexisting lower-level components; exploiting the preexisting to generate significant and viable phenotypic variation is entirely consistent with facilitated variation [10, ch. 7]. The fact that no similar stasis points occur for 11-mux(R) in figure 5, the ROBDD of which does not make extensive reuse of subfunction, further suggests punctuated evolution as a possible side affect of this kind of modularity.

5 Focussing Mutation: A Thought Experiment

A thought experiment will elucidate the mechanism of EMMF. The target function for the experiment is the 11-mux, but any function with a compact ROBDD representation would suffice. The variable ordering is static.

Using only the N1' and N2' mutation operators, any ROBDD can be expanded so that the nonterminals form a tree in the nonterminals. The tree representation of 11-mux has size exponential in the number of variables in contrast to the ROBDD representation of 11-mux, which is linear in size. Clearly, there are a plethora of OBDD representations intermediate in size between the tree and ROBDD representations of 11-mux. In general, the larger the OBDD representation, the greater the level of redundancy and the lower the level of modularity. Thus, the tree representation of 11-mux has the highest level of redundancy, with many redundant subfunctions. It is the fact that the tree representing 11-mux has high regularity of subfunctions that allows the tree to be compressed to a maximally modular ROBDD having linear complexity.

Now, consider the effect of random mutation, without selection, on the tree representation of 11-mux. The mutation operators are both neutral and non-neutral and applied to random locations repeatedly. Further, assume that the size of the OBDD remains similar to the tree. Clearly, the function represented by the OBDD will change from 11-mux. However, and more importantly, mutation will disrupt the regularity that was present before: it must be expected that a greater variety of subfunctions will result. The consequence will be that the mutated OBDD will be less compressible than 11-mux and have a larger ROBDD representation.

Now consider the tree representation of 11-mux again. This time, however, mutation is applied only to one side of the tree; the other half is left untouched. Clearly, the untouched half will retain both its function and compressibility, but the mutated half will retain neither. The representation of the functional part of the OBDD that is correct can be compressed into a comparatively small

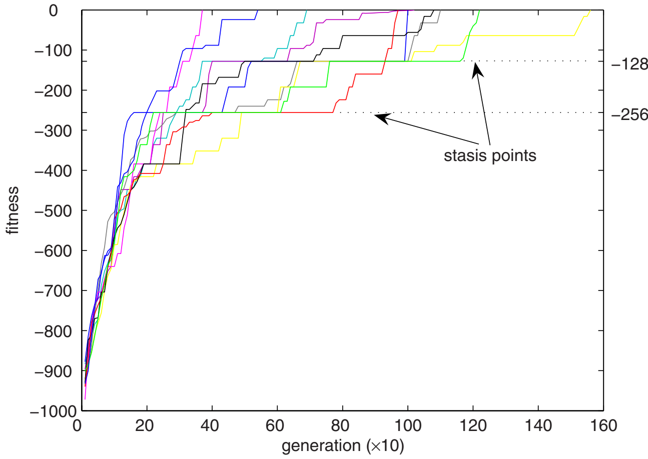


Fig. 4. Fitness curves for 11-mux exhibiting punctuated equilibria-like characteristics. 10 curves are shown, most of which get exhibit periods of stasis at the points indicated. The stasis points are located at points 2^n, implying the absence of a higher-level functional component. All runs eventually return to rapid fitness improvement once the foundations of the absent component are discovered, exploiting lower-level components.

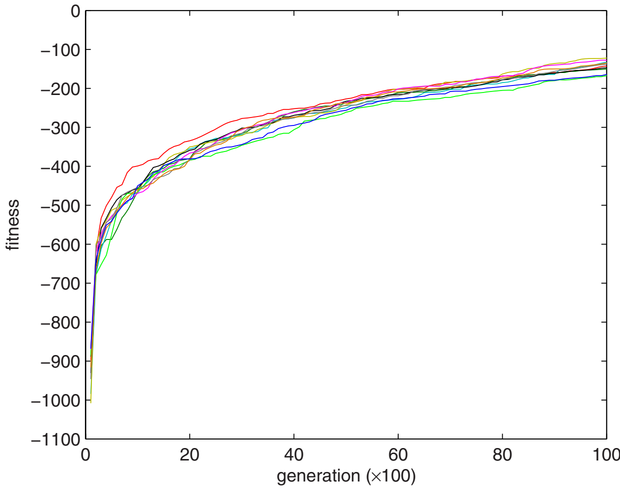


Fig. 5. Fitness curves for 11-mux(R). 10 curves are shown. Fitness improvement is gradual with no significant periods of stasis. Lower-level components are not being exploited by higher-level components. These curves are not indicative of punctuated evolution.

structure, while the part that is not functionally correct will have a comparatively larger representation under compression. That part of the genotype that represents the incorrect part of the function now presents a relatively bigger target for mutation. Thus, the part of function that is incorrect attracts a disproportionate amount of applications of the A1 operator under random mutation. That part of the function that is correct is protected from mutation by the relative density of its representation.

The situation described above could not arise under normal evolutionary forces: it simply serves as an example to illustrate that the representation of the correct part of the function is compressible, while the representation of the randomised part is not, and the consequence this has for biasing mutation. The question is whether this effect can arise under normal evolutionary forces? To address this question, the combined effects of all the mutation operators and selection must be considered over several generations. It will assist to group the effects of mutation into three principal forces.

- *Disruptive force.* The expanding neutral mutations, N1' and N2', combined with the non-neutral A1 mutation serves to decompress and disrupt regularity, inhibiting compression.
- *Compressive force.* Reducing neutral mutations, N1 and N2, serve to compress parts of the representation, regardless of whether those parts are adapted.
- *Preservative force.* Selection serves to preserve the adapted part of the function, propagating it to future generations.

Expanding parts of the genotype that represent adapted function exposes that part to disruption by A1. Given that disruption of correct function will usually result in deleterious offspring, such lineages will be selected against. This is nothing more than stabilising selection preferring those configurations that less expose adapted functionality to mutation. Expanding parts of the genotype that represent maladapted function, however, will greater expose the maladapted function to disruption by A1, which will serve to improve fitness and inhibit compression of the representation of the maladapted function, and will not be selected against. Preexisting or newly adapted function that is uncompressed is compressible using N1 and N2, while maladapted function is less so. That maladapted function that is compressible is relatively more susceptible to expansion and functional disruption, and so the cycle continues.

6 The Genotypic Distribution of Functionality

This section investigates the hypothesised emergence of such a biased distribution. In order to do so, the concept of *utility* is introduced. Utility provides an indication of the relevance of difference parts of the genotype to the adapted as opposed to maladapted functionality. The parts of the genotype that are of primary interest are the edges rather than the vertices, as it is the edges that are redirected under A1 mutation. However, it may also make sense to talk about

the utility of a vertex or other genotypic features in some contexts. The *utility density*, $\mathcal{U}_d$, is the total number of fitness cases processed by an edge. The *utility proportion*, U_p, is the fraction of fitness cases processed by an edge that are correct. The *expected utility proportion*, U_E, is the fraction of all fitness cases used in evaluation that are correct. Clearly, U_E correlates with fitness.

The root vertex processes all fitness cases, so $U_p = U_E$ here. In any randomly generated OBDD genotype, $U_p \approx U_E$ should be exhibited throughout the genotype: U_p not approximating U_E indicates a biased distribution of the representation of adapted and maladapted functionality. EMMF predicts a distortion of the distribution of U_p under the forces of random mutation and selection when the target has a compact ROBDD representation. Any bias should distort the bulk of the distribution away from U_E towards the extremes $U_p = 1$ and $U_p = 0$, the former becoming more prominent as fitness increases. 11-mux(R) has the reverse variable ordering, and an ROBDD having exponential complexity in contrast to the linear complexity of 11-mux. Therefore, while 11-mux is subject to EMMF, 11-mux(R) should be less so, and should exhibit a distribution of U_p more consistent with U_E.

7 Analysis

Figure 6 contrasts frequency polygons for 11-mux and 11-mux(R). Each frequency polygon has 20 buckets with centres in the range 2.5 through 97.5 in steps of 5, representing frequencies of U_p as a percentage. The frequency polygon connects the top centres of the histogram, and can facilitate easier comparison of distributions than a histogram. There are five sub figures, each representing intervals of fitness as U_E in the range indicated by the vertical bars. Samples are taken of the parent at each fitness improvement step and averaged over the interval, though the distributions were found not to be significantly different at other times also. To accommodated genotypes of varying size, U_p frequencies are normalised for each individual so that the sum of frequencies is 1.

The shape of the distributions are telling. 11-mux(R) approximates a peaked skewed distribution of U_p with a mode close to U_E. This indicates that the distribution of functionality is very close to that which would result from a randomly generated OBDD representation of the function. However, 11-mux has a very different distribution. As fitness increases the distribution spreads considerably from U_E, with a much higher proportion of edges exhibiting maximum values of U_p, or values much less that U_E. The distribution begins to form a hollow at U_E and the upper tail soon disappears before optimal fitness is neared, implying a very biased distribution of functionality within the genotype.

In both cases, the majority of lower values of U_p are quickly eliminated. An edge with a low value of U_p, say $U_p <= 50\%$, has a good probability of providing a fitness improvement under A1 mutation because mutating that edge to a random subfunction has and expected U_p of around 50%. However, given that good subfunctions must be present in the genotype for above average fitness to be exhibited, it can be expected that the result of mutating an edge with A1 will

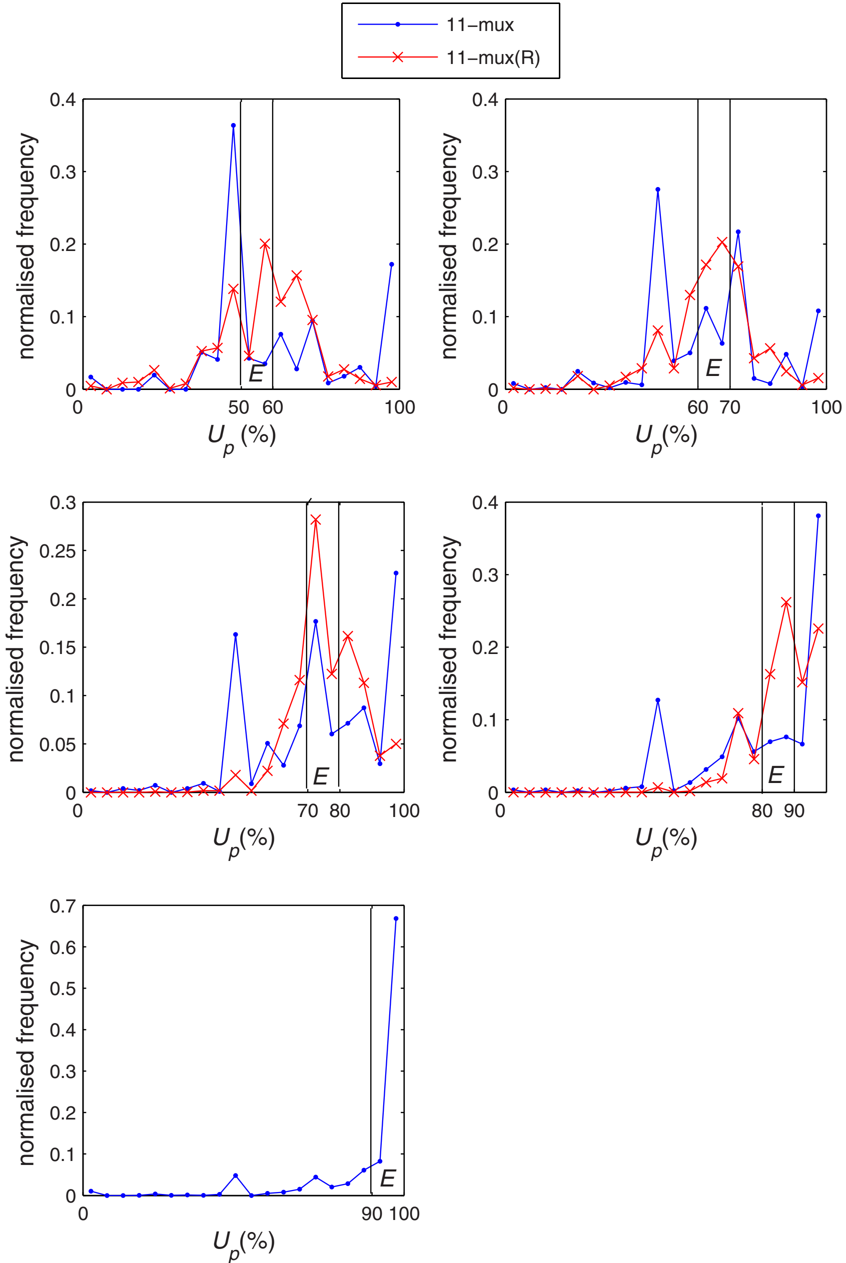


Fig. 6. Frequency distributions of U_p for 11-mux and 11-mux(R). Each plot shows an interval of fitness as U_E. No plot is shown for 11-mux(R) in the interval 90-100% as fitness improvement stagnated here. The distribution of U_p for 11-mux(R) approximates U_E. The distribution of U_p for 11-mux, however, sees a biased distribution in which the upper tail disappears completely and the lower tail is extended.

result in $U_p > 50\%$ for mux-11. Still, the figure shows that small frequencies of $U_p <= 50\%$ persist up into the highest fitness interval for 11-mux, exhibited as an extended lower tail on the distribution.

Similar experiments were conducted on both the parity and adder functions. The results here were consistent with the EMMF hypothesis also, giving further confidence. The 10-bit parity is insensitive to changes in π, but the bias was similar to that for 11-mux depicted in figure 6. For the 4-bit adder, both best and worst π were compared. The former exhibited a bigger bias, and the latter had a lesser bias not consistent with U_E. However the growth in ROBDD complexity for the worst π is not as severe to approximate a tree, allowing some room for compression. It is the comparison of the distributions of the worst and best π for a function that is telling.

8 Conclusion

The EMMF hypothesis predicts an emergent bias in the distribution of the representation of adapted and maladapted functionality within the genotype, and this has been born out by experimentation. Where the target function has the potential for modularity (i.e. a compact ROBDD), mutation and selection give rise to the bias. Where the target function does not have the potential for modularity, no bias will emerge. Downing [8] has shown that better π emerge as a logical consequence of being correlated with evolvability under EBDDIN with a dynamic π. This implies that the emergence of better π better facilitates the bias. Thus, a cascading influence is imparted that induces the representation to evolve at different levels of organisation to facilitate evolvability.

Some of the properties of EBDDIN that facilitate EMMF are important to appreciate. The *massive redundancy* of the OBDD representation facilitates a plethora neutral genotypic configurations to explore, each offering different distrubtion in the representation of adapted and maladapted functionality. Without this redundancy, genotypes exhibiting a biased distribution of functionality would not occur so readily. The *neutral networks* that connect all the representations of a given function facilitates exploration of those configurations, allowing the favourable configurations to emerge by the reproductive advantage they impart. The neutral evolution of modularity towards favourable configurations is necessarily *gradual*, else favourable configurations would be difficult to maintain within the population and neutral walk would be stifled.

It was shown that evolution of a modular target can give rise to punctuated evolutionary characteristics. This was suggested as a result of delayed discovery of foundations of significant functional components. Once such a foundation is discovered, however, evolution continues apace, exploiting lower-level components and providing significant, but viable, phenotypic change. This is also consistent with facilitated variation [10], which postulates the means by which minimal genotypic mutation can trigger significant, but viable, phenotypic change (e.g. a second set of insect wings [10, p. 190]).

Future work will seek to further understand evolvability issues in EBDDIN and other representations. A deeper understanding of evolvability, its emergence and the mechanisms that facilitate it will aid in the design of new EAs that address scalability issues for application-specific problems.

References

1. Akers, S.B.: Binary Decision Diagrams. IEEE Transactions on Computers C-27(6), 509–516 (1978)
2. Altenberg, L.: The evolution of evolvability in genetic programming. In: Kinnear Jr, K.E. (ed.) Advances in Genetic Programming, ch. 3, pp. 47–74. MIT Press, Cambridge (1994)
3. Henrik Reif Andersen. An introduction to Binary Decision Diagrams. Notes for course 49285: Advanced Algorithms, Technical University of Denmark (1997)
4. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming: An Introduction. Morgan Kaufmann, San Francisco (1998)
5. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Transactions on Computers 45(9), 993–1002 (1996)
6. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers C-35(8), 677–691 (1986)
7. Downing, R.M.: Evolving Binary Decision Diagrams using implicit neutrality. In: Corne, D., Michalewicz, Z., Dorigo, M., Eiben, G., Fogel, D., Fonseca, C., Greenwood, G., Chen, T.K., Raidl, G., Zalzala, A., Lucas, S., Paechter, B., Willies, J., Guervos, J.J.M., Eberbach, E., McKay, B., Channon, A., Tiwari, A., Volkert, L.G., Ashlock, D., Schoenauer, M. (eds.) Proceedings of the 2005 IEEE Congress on Evolutionary Computation, September 2-5, vol. 3, pp. 2107–2113. IEEE Press, Los Alamitos (2005)
8. Downing, R.M.: Evolving binary decision diagrams with emergent variable orderings. In: Runarsson, T.P., Beyer, H.-G., Burke, E.K., Merelo-Guervós, J.J., Whitley, L.D., Yao, X. (eds.) PPSN 2006. LNCS, vol. 4193, pp. 798–807. Springer, Heidelberg (2006)
9. Gould, S., Eldredge, N.: Punctuated equilibrium comes of age. Nature 366, 223–227 (1993)
10. Kirschner, M.W., Gerhart, J.C.: The Plausibility of Life: Resolving Darwin's Dilemma. Yale (2005)
11. Lee, C.Y.: Representation of Switching Circuits by Binary-Decision Programs. Bell Systems Technical Journal 38, 985–999 (1959)
12. Shannon, C.E.: A symbolic analysis of relay and switching circuits. Transactions of AIEE 57, 713–723 (1938)
13. Sieling, D.: On the existence of polynomial time approximation schemes for OBDD-Minimization. In: Meinel, C., Morvan, M. (eds.) STACS 1998. LNCS, vol. 1373, pp. 205–215. Springer, Heidelberg (1998)
14. Woodward, J.R.: Modularity in genetic programming. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E.P.K., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 258–267. Springer, Heidelberg (2003)

A Linear Estimation-of-Distribution GP System

Riccardo Poli[1] and Nicholas F. McPhee[2]

[1] Department of Computing and Electronic Systems, University of Essex, UK
rpoli@essex.ac.uk
[2] Division of Science and Mathematics, University of Minnesota, Morris, USA
mcphee@morris.umn.edu

Abstract. We present N-gram GP, an estimation of distribution algorithm for the evolution of linear computer programs. The algorithm learns and samples a joint probability distribution of triplets of instructions (or 3-grams) at the same time as it is learning and sampling a program length distribution. We have tested N-gram GP on symbolic regressions problems where the target function is a polynomial of up to degree 12 and lawn-mower problems with lawn sizes of up to 12×12. Results show that the algorithm is effective and scales better on these problems than either linear GP or simple stochastic hill-climbing.

1 Introduction

Estimation of distribution algorithms (EDAs) (see [4] for a review) are powerful population-based searchers where the variation operations traditionally implemented via crossover and mutation in evolutionary algorithms are replaced by the process of sampling from a distribution. For example, PBIL [2] and UMDA [6] assume that the distribution is a product of univariate marginals. EDAs modify the distribution generation after generation, using information obtained from the more fit individuals in the population. The objective of these changes is to increase the probability of generating individuals with high fitness. Different algorithms use different models for the probability distribution that controls the sampling.

There have been several applications of EDA-style probabilistic model-based evolution to tree-based GP; we review several below and provide a full literature review in [8]. In PIPE [11], the first EDA-type GP system, the population is replaced by a hierarchy of probability tables organised into a tree, where each table represents the probability that a particular instruction be at that particular location in a newly generated program tree. eCGP [12] assumes that all trees will be created by sampling within a maximal tree and partitions the nodes in this tree into groups. The co-occurrence of the nodes in each group is modelled by a full joint distribution table. EDP [16] uses a conditional probability table which can, in principle, capture more complex dependencies between nodes. As in eCGP and PIPE, programs are tree-like and are assumed to always fit within an ideal maximal full tree. A hybrid between EDP and GP was proposed in [17].

Various other systems have been proposed which combine the use of grammars and probabilities.[1] For example, [10] used a stochastic context free grammar to generate

[1] In fact, there is a fundamental equivalence between probabilistic grammars and other probabilistic approaches (see [14] for a detailed explanation).

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 206–217, 2008.
© Springer-Verlag Berlin Heidelberg 2008

program trees. The probability of application of each rewrite rule was adapted using a standard EDA approach so as to increase the probability of application of successful rules. Slightly more general is the approach taken in PEEL (Program Evolution with Explicit Learning) [13], where a probabilistic L-system is used with rewrite rules that are depth- *and* location-dependent and have associated probabilities of application which are adapted by an Ant Colony Optimisation (ACO) algorithm. Another programming system based on a probabilistic grammar optimised via ant systems is ant-TAG [1].

While there is a significant amount of prior work, there has been no application of EDA-style ideas to linear GP. This paper starts filling this gap by proposing *N-gram GP*, an EDA-type GP system capable of evolving machine-language-type programs [7]. A further novelty is our use of n-grams, borrowed from the field of natural language processing, to represent regularities in the language necessary to solve a problem.

2 N-Gram GP

An n-gram is a group of n consecutive items from a longer sequence. For example, `a b`, `b c` and `c d` are all 2-grams from the sequence `a b c d`, while `a b c` and `b c d` are 3-grams. The items in the sequence can be of a variety of types, including words from natural language, base pairs in a DNA fragment, and phonemes in a speech recording. Very often n-grams are used for the purpose of modelling the statistical properties of sequences, particularly natural language [15,9,5]. In particular, an n-gram model assumes that the probability of a particular symbol appearing in a sequence depends only on what appeared before that symbol and in its vicinity in the sequence. More formally, if we imagine that a particular sequence $x_1, x_2, \ldots$ is an instantiation of a family of stochastic variables $X_1, X_2, \ldots$, the assumption is that, for any k-gram, $\Pr\{X_i = x_i | X_{i-1} = x_{i-1}, \ldots, X_{i-k+1} = x_{i-k+1}\}$ is independent of i and is sufficient to correctly capture the probability of X_i taking the value x_i in a particular sequence.

In this work we will use an n-gram distribution to generate linear computer programs, that is sequences of instructions from the assembly language of a register-based CPU, as in linear GP [7]. We will limit our attention to the case of 3-grams. So, if $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$ is the primitive set, our model of the language can then be represented by a matrix $M^{(3)} = (m_{l,m,n})$ with elements $m_{l,m,n} = \Pr\{X_i = p_n, X_{i-1} = p_m, X_{i-2} = p_l\}$, where the indices l, m, and n range over the set $\{1, \ldots, N\}$. From this matrix we derive two further matrices, $M^{(2)} = (m_{l,m})$ and $M^{(1)} = (m_l)$, with elements $m_{l,m} = \sum_n m_{l,m,n}$ and $m_l = \sum_m m_{l,m}$, respectively. So, the elements of these matrices are marginals of the distribution $\Pr\{X_i = p_n, X_{i-1} = p_m, X_{i-2} = p_l\}$.

Note that, in general, by definition $\Pr\{X_i = x_i, X_{i-1} = x_{i-1}, \ldots, X_{i-k+1} = x_{i-k+1}\}$ $= \Pr\{X_i = x_i | X_{i-1} = x_{i-1}, \ldots, X_{i-k+1} = x_{i-k+1}\} \times \Pr\{X_{i-1} = x_{i-1}, \ldots, X_{i-k+1} = x_{i-k+1}\}$. So, the elements of $M^{(3)}$ are proportional to $\Pr\{X_i = p_n | X_{i-1} = p_m, X_{i-2} = p_l\}$, the elements of $M^{(2)}$ are proportional to $\Pr\{X_{i-1} = p_m | X_{i-2} = p_l\}$, and, finally, the elements of $M^{(1)}$ are proportional to the priors $\Pr\{X_{i-2} = p_l\}$.

Assuming that the matrices $M^{(1)}$, $M^{(2)}$ and $M^{(3)}$ are available, one can then use them to generate sequences of instructions with the same statistical properties as the language the n-gram model is supposed to represent. The process starts by sampling the primitive set $\mathcal{P}$ using probability distribution $M^{(1)}$ to obtain the first instruction of a program.

The second instruction is drawn by using the distribution prescribed by the row of $M^{(2)}$ corresponding to the first primitive. The third and all subsequent instructions are then drawn using the appropriate entries from $M^{(3)}$. (See Alg. 1 for more details.)

How does this process terminate? If one of the instructions in the primitive set $\mathcal{P}$ is an `EXIT` instruction of some type, the construction process can naturally terminate when such an instruction is drawn. The program length distribution is then determined by the probability of drawing `EXIT` instructions. An alternative, which gives more control on program length, is to dispense with the `EXIT` instruction and instead explicitly represent and update a program length distribution. In this case a length is first drawn from the length distribution, and programs are then grown up to the prescribed length. In this paper we take the latter approach.

One option when explicitly representing the length would be to add an extra dimension to our N-gram model, making it possible to learn different 3-gram distributions for different length classes. However, this is problematic. The matrix $M^{(3)}$ is of size N^3, where N is the size of the primitive set. Extending the model to 4 dimensions implies an increase in the number of parameters to be learnt by two or more orders of magnitude, which is a distinct disadvantage. So, we decided in this initial research to focus on a system where program length is independent from program primitives. That is, we assume that the joint distribution of 3-grams and length is a product of the form $M^{(3)} \times P_L$, where P_L is the length distribution. The construction of new programs, therefore, proceeds as shown in Alg. 1.

So far we have assumed that the distributions $M^{(3)}$ and P_L were somehow available. Now we look at how we construct such models. We do this using a standard EDA approach with minor modifications as shown in Alg. 2. We start by initialising the distributions P_L and $M^{(3)}$. If we have no prior information on the problem to be solved (as is assumed in all experiments reported in the paper), the most natural initialisation is the uniform distribution. If ℓ_{max} represents the maximum program size we are interested in, then all entries of P_L are initialised to $1/\ell_{max}$. Similarly, all the entries of $M^{(3)}$ are initialised to $1/N^3$. Then, after projecting $M^{(3)}$ to obtain its marginals, we proceed to construct a new population. This is, for the most part, created by sampling from our model (the distribution $M^{(3)} \times P_L$). However, occasionally (on average once per generation) the best individual seen so far in the run is reintroduced to guarantee stability in the estimated distribution. To maintain diversity and ensure that the search continues even after many entries of $M^{(3)}$ converge to 0, we follow standard EDA practice and perform point mutation (at a low per-locus rate) on the individuals returned by the `genProgram` routine. Like in many EDAs, the population then undergoes a step of truncation selection, where the best individuals are stored in a set `elite` which is then used to update the program distribution; in this work we used the top 1/5 of the population for `elite`.

The update of the distribution is performed independently for P_L and $M^{(3)}$ using an additive update rule as shown in Alg. 3. Note that the arrays are not explicitly zeroed before they are updated. In this way the model used to produce individuals at one particular generation can depend also on successful individuals discovered in previous generations in the run. How much the current `elite` influences the model depends on two learning rates, η_M and η_L. If desired, these can be made arbitrarily big. When $\eta_M \gg 1$

Algorithm 1. Program generation algorithm for N-gram GP.

genProgram($M^{(1)}$**,** $M^{(2)}$**,** $M^{(3)}$**,** P_L**)**

1: Select program length, $\ell > 0$ based on the probabilities stored in the distribution P_L {Perform a roulette wheel selection on the entries of P_L}
2: Select the first instruction, x_1, based on the probabilities stored in $M^{(1)}$ {via roulette wheel}
3: If $\ell > 1$ select the second instruction, x_2, based on the probabilities stored in the x_1-th row of $M^{(2)}$, which we indicate with $M^{(2)}_{x_1}$ {Again this is done via roulette wheel selection on $M^{(2)}_{x_1}$}

4: **for** $i = 3$ **to** ℓ **do**
5: Select x_i based on $M^{(3)}_{x_{i-2},x_{i-1}}$ {$M^{(3)}_{x_{i-2},x_{i-1}}$ is the x_{i-2}-th row in the x_{i-1}-th page of $M^{(3)}$}
6: **end for**
7: **return** $x_1, x_2, \cdots, x_\ell$

Algorithm 2. N-gram GP main loop.

N-gram-GP

1: Initialise the distributions P_L and $M^{(3)}$
2: **repeat**
3: Compute marginals of $M^{(3)}$ to obtain $M^{(1)}$ and $M^{(2)}$
4: **for** $i = 1 \ldots$ `popsize` **do**
5: With probability $1/$`popsize`, `pop[`i`]` = best individual found so far
6: With probability $1 - 1/$`popsize`, `pop[`i`]=mutate(genProgram(`$M^{(1)}$, $M^{(2)}$, $M^{(3)}$, P_L`))`
7: **end for**
8: `elite = truncationSelection( pop )`
9: `updateProbabilities(` P_L, $M^{(3)}$, `elite )`
10: **until** Solution found or max number of iterations exhausted
11: **return** best individual found

and $\eta_L \gg 1$, P_L and $M^{(3)}$ are almost entirely determined by the current elite, effectively independent of the previous history of the run.

3 Experimental Results

3.1 Problems and Primitive Sets

We used two families of test problems: `Polynomial` and `Lawn-Mower`. `Polynomial` is a symbolic regression problem where the objective is to evolve a function which fits a polynomial of the form $x + x^2 + \cdots + x^d$, where d is the degree of the polynomial, and x is in the range $[-1, 1]$. In particular we considered degrees $d = 5, \ldots, 12$, and we sampled the polynomials at the 21 equally spaced points $x \in \{-1.0, -0.9, \ldots, 0.9, 1.0\}$. Fitness (to be minimised) was the sum of the absolute differences between target polynomial and the output produced by the program under evaluation over these 21 fitness cases. Polynomials of this type have been widely used as benchmark problems in the GP literature. However, we are unaware of any experiments with degrees as high as the ones we consider here.

Algorithm 3. Learning in N-gram GP.

updateProbabilities(P_L, $M^{(3)}$, `elite` **)**

1: **for all** x **in** `elite` **do**
2: $\ell =$ `length(` x `)`
3: $P_{L,\ell} = P_{L,\ell} + \eta_L / \ell_{max}$
4: **for** $j = 3 \ldots \ell$ **do**
5: $M^{(3)}_{x_{j-2},x_{j-1},x_j} = M^{(3)}_{x_{j-2},x_{j-1},x_j} + \eta_M / N^3$
6: **end for**
7: **end for**
8: $M^{(3)} = M^{(3)} / \sum_{l,m,n} M^{(3)}_{l,m,n}$
9: $P_L = P_L / \sum_l P_{L,l}$.

Table 1. Primitive sets used in our experiments (% represents protected division, which returns its first argument if the second argument is zero)

	Polynomial		Lawn Mower
ID	PlusTimesSwapR1R2	AllOpsSwapR1R2	
0	R1 = RIN	R1 = RIN	Mow
1	R2 = RIN	R2 = RIN	Left
2	R1 = R1 + R2	R1 = R1 + R2	Right
3	R2 = R1 + R2	R2 = R1 + R2	
4	R1 = R1 * R2	R1 = R1 * R2	
5	R2 = R1 * R2	R2 = R1 * R2	
6	Swap R1 R2	Swap R1 R2	
7		R1 = R1 - R2	
8		R2 = R1 - R2	
9		R1 = R1 % R2	
10		R2 = R1 % R2	

For these problems we considered two primitive sets: `PlusTimesSwapR1R2`, that is particularly suitable for the solution of this problem, and `AllOpsSwapR1R2` which is a superset of `PlusTimesSwapR1R2` containing two spurious primitives. These primitive sets are detailed in the first two columns of Table 1. The instructions refer to three registers: the input register `RIN` which is loaded with the value of x before a fitness case is evaluated and the two registers `R1` and `R2` which can be used for numerical calculations. `R1` and `R2` are initialised to x and 0, respectively. The output of the program is read from `R1` at the end of its execution.

`Lawn-Mower` is a variant of the classical Lawn Mower problem introduced by Koza in [3]. As in the original version of the problem, we are given a square lawn made up of grass tiles. In particular, we considered lawns of size $d \times d$ with $d = 5, \ldots, 12$. The objective is to evolve a program which allows a robotic lawnmower to mow all the grass. In our version of the problem, at each time step the robot can only perform one of three actions (see Table 1): move forward one step and mow the tile it lands on (`Mow`), turn left by 90 degrees (`Left`) or turn right by 90 degrees (`Right`). In the original problem fitness (to be minimised) was measured by the number of tiles left non-mowed at the

end of the execution of a program; whether or not the lawnmower kept visiting other tiles after finishing the job was not considered a relevant part of the problem. This makes the problem rather easy to solve and uninteresting if the GP system is allowed to grow large enough programs. So, to make the problem more difficult, we implemented two further constraints. First, we limited the number of instructions allowed in a program to a small multiple of the number of tiles available in the lawn (more precisely $4 \times d^2$). Second, we required the lawnmower to be energy efficient, so we added corrections to the fitness function which encouraged the evolution of rapidly-mowing programs and programs that stop immediately after having cut the last grass patch:

$$\text{fitness} = \begin{cases} 0.0001 \times \text{extraMoves} & \text{if all tiles mowed} \\ 0.1 \times \text{progLength} + \text{numUnmowedTiles} & \text{otherwise} \end{cases}$$

where *extraMoves* is the number of moves made after the last tile was mowed.

3.2 Other Algorithms Used for Comparison

In order to evaluate the strengths and weaknesses of N-gram GP, we tested it against two other techniques: simple stochastic hill climbing and a traditional linear GP system. All algorithms were given the same number of fitness evaluations (20,000 in all experiments reported in the paper) and were individually optimised (by doing a large sweep of their parameter space) to maximise their performance on our test problems. These parameters are detailed in Table 2.

The hill climber is initialised by choosing a random program length between 1 and ℓ_{max} and then generating a random program of that length. From then on, the algorithm repeatedly attempts to improve over the best individual seen so far by randomly mutating it. The algorithm has two equally probable mutation operations to choose from. The first is point mutation which is applied to the primitives of the best program with a mutation rate of p/ℓ where ℓ is the length of the current best program and p is a parameter ($p = 2$ in our experiments). This form of mutation cannot change program length. The second form of mutation is effectively subtree mutation applied to linear sequences. I.e., a random mutation point is chosen in the parent individual, all of the instructions following the mutation point are excised, and they are replaced by a newly generated random sequence. As a result, the offspring program can have a length which is different from the parental length.

Our linear GP system works as follows. It initialises the population by repeatedly creating random individuals using the same distribution as for the initialisation of the hill climber, and evaluating their fitness. Then a typical steady state evolutionary loop starts. At each iteration the algorithm decides whether to create a new individual via mutation or crossover. If mutation is chosen, a parent individual is selected via tournament selection (with tournament size 2), and an offspring is generated using the same algorithm as for the hill climber (i.e., we use point mutation 50% of the time, and subtree mutation the other 50% of the time). If crossover is chosen, we select two parents (again, via tournament selection) and apply homologous two-point crossover with 50% probability, and subtree crossover with 50% probability. Subtree crossover involves the selection of one crossover point in each parent, and the swap of the instructions following the crossover

Table 2. Parameter settings for hill-climber, linear GP and N-gram GP

Parameter	*Hill-Climber*	*Linear GP*	*N-gram GP*
Fitness evaluations	20,000	20,000	20,000
Independent runs	1,000	1,000	1,000
ℓ_{max} Polynomial Problem	100	100	100
Lawn-Mower	$4 \times d^2$	$4 \times d^2$	$4 \times d^2$
Point mutation rate (per primitive)	$\frac{2}{\ell}$	$\frac{1}{\ell}$	$\frac{0.25}{\ell}$
Population size	1	500	10
Generations	20,000	40	2,000
Crossover rate (per individual)	n/a	0.9	n/a
Mutation rate (per individual)	1	0.1	1
Tournament size	n/a	2	n/a
Truncation selection ratio	n/a	n/a	5
η_M learning rate	n/a	n/a	8
η_L learning rate	n/a	n/a	0.075

points. Homologous crossover requires choosing the same crossover points in both parents. Irrespective of the genetic operation chosen, the individual picked for replacement is selected via a negative tournament.

3.3 Performance Results

Fig. 1(a) shows a comparison of the success rate obtained by the hill climber, the N-gram GP and the linear GP on Polynomial problems of degrees from 5 to 12, when using the `PlusTimesSwapR1R2` primitive set. It is apparent how the use of a small set of suitable primitives makes the problem solvable for all techniques tested. Unsurprisingly, the simple hill-climber does well on the relatively easy instances, but its performance is unsatisfactory for d bigger than 9 or 10. On the contrary, the performance of the linear GP never really drops to unacceptable levels. Furthermore, both linear GP and the hill-climber do marginally better than the N-gram GP for small d. On the more difficult problems, however, N-gram GP shows much better performance. Furthermore, its performance drops more slowly than the other techniques as d increases, suggesting better scalability on these problems.

As illustrated in Fig. 1(b), the use of a larger-than-necessary primitive set (`AllOpsSwapR1R2`) makes the problem harder, because the size of the search space increases enormously without a corresponding increase in the size of the solution space. In these conditions, a searcher would need more time and resources to identify solutions. However, in this work, all problems, searchers and primitive sets are compared using the same number of fitness evaluations, leading to generally poorer performance. Despite this general trend, all observations we made for Fig. 1(a) appear to be valid for Fig. 1(b) as well. In particular the N-gram GP system still shows superior scalability and higher performance on the harder problems.

Finally, let us consider the `Lawn-Mower` problem. Again the simple hill-climber is the weakest of all searchers. Linear GP is marginally superior to N-gram GP for small problem sizes. However, its performance rapidly degrades, becoming unacceptable for

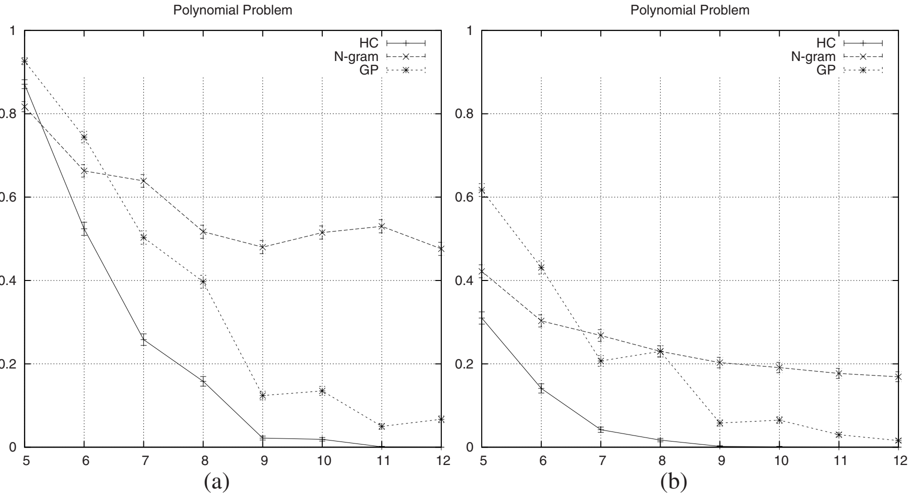


Fig. 1. Success rate of hill climber, N-gram GP and linear GP on Polynomial problems of degrees from 5 to 12, when using the `PlusTimesSwapR1R2` primitive set (a) and the `AllOpsSwapR1R2` primitive set (b). Parameters are as detailed in Table 2.

problems of size $d = 11$ or larger. Conversely, N-gram GP's performance degrades much more gracefully, leading it to being still able to solve problems of size $d = 12$ in about one third of runs.

4 Analysis and Discussion

The results from Sec. 3.3 clearly show that N-gram GP outperforms hill-climbing and often also linear GP, so the N-gram GP is obviously learning during a run. What is being learned, and how is that happening? Presumably, if there is learning going on, it is in the proportions being stored in the $M^{(i)}$ and P_L arrays. To better understand this learning, we will examine the kinds of bias that develops in these matrices in successful runs.

Of particular interest is whether (and how) N-gram GP learns longer patterns when limited to only keeping statistics on the occurrence of triplets. Even though N-gram GP can only learn triplets, there can obviously be correlations across multiple triplets; if *wxy* and *xyz* both have high probabilities, then the 4-tuple *wxyz* is a likely outcome if one starts with *wx*. To what degree does N-gram GP utilise this opportunity? How long are the patterns that it learns? What role does repetition play in those patterns?

To address these questions, we took successful runs, generated probability trees based on the distribution matrices, and catalogued programs that were generated with high probabilities. In every instance that we explored, there were clear patterns of instructions captured in the distribution matrices. In Sec. 3.3, the runs were halted once the goal was discovered. The distribution matrices often were not strongly converged when the goal was first found, so for this section we allowed runs to continue through the maximum allowed number of fitness evaluations, regardless of whether a solution was discovered.

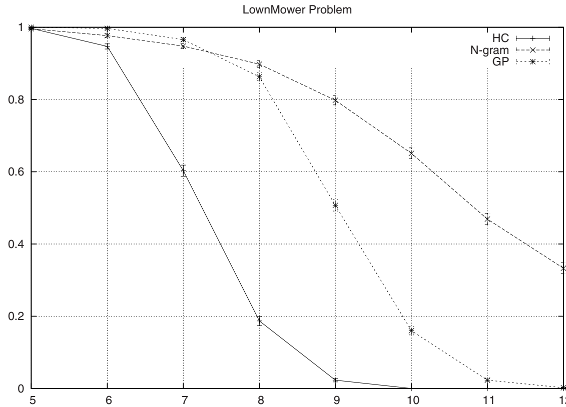


Fig. 2. Success rate of hill climber, N-gram GP and linear GP on Lawn-Mower problems with lawn sizes from 5×5 to 12×12. Parameters are as detailed in Table 2.

This gave the distribution matrices the opportunity to continue converging after finding the goal, making it easier to see what was being learned. To make it easier to present long sequences of instructions, and to see patterns in those sequences, we will present programs as sequences of the integer IDs of the instructions, using the IDs given in Table 1.

4.1 Polynomial

To simplify our analysis of the polynomial problems, we will consider each sequence of instructions as a mapping from one state of the system to another. Since the state in the regression problems is fully determined by the contents of the registers, we can represent that state as an ordered tuple. In the two register problems, for example, we can use an ordered pair (r_1, r_2) to represent the values of `R1` and `R2`, respectively. Consider, for example, the pair of instructions represented by the index sequence 53, namely instruction 5 (`R2 = R1 * R2`) followed by instruction 3 (`R2 = R1 + R2`). If we start the system in state (a,b) (where a and b represent arbitrary initial values) and execute this pair of instructions we end in the state $(a, a(b+1))$.

It turns out that polynomials of the form $x + x^2 + \cdots + x^d$ can be easily constructed in a highly patterned way using our simple instruction set. In one run, for example, with the degree 7 target $(x + x^2 + \cdots + x^7)$, the evolved distribution matrices have a high probability of generating sequences containing repetitions of the instruction pair 53, such as the solution 1535353535352. Here the initial instruction (1) loads x into R2, which, because `R1` is initialised to contain x, means that our state is (x,x) after that first instruction. The first pair 53 then maps this to $(x, x + x^2)$, the second to $(x, x + x^2 + x^3)$, and so on until the last 53 pair yields the state $(x, x + x^2 + x^3 + x^4 + x^5 + x^6)$. The final pair is 52 which has a similar effect, but leaves the result in `R1`, so we have the final state $(x + x^2 + \cdots + x^7, x^2 + x^3 + x^4 + x^5 + x^7)$, which has our target function in `R1`.

This solution is actually somewhat brittle because it depends crucially on getting the final 52 pair at the end, while having 53's everywhere else. This is reflected in the

probability matrices, where the probability of following 35 with a 3 is 66%, while the probability of following 35 with a 2 is only 1.3%. Contrast this with the probabilities following the pair 53, where the probability of a 5 is greater than 99.999%. Consequently the system has learned that the sequence 53 should *always* be followed by a 5, where the sequence 35 should *usually* be followed by 3, but occasionally by a 2 instead.

A somewhat more robust solution found on another run is 3412412412412412412. Here the only "novelty" is the initial 3, which ensures that both registers have a copy of the argument x at the beginning. Then the pattern 412 generates the sequence of polynomials $x, x+x^2, x+x^2+x^3, \ldots, x+x^2+x^3+x^4+x^5+x^6+x^7$ in `R1`. This solution is also interesting in that it in fact generates *every* polynomial of the form $x+x^2+\cdots+x^d$, and all that is necessary to generate polynomials of other degrees is to increase or decrease the number of 412 triplets appropriately.

One might reasonably assume that the probability of generating the sequence 412 must be very high in this solution, perhaps approaching 1. It is, however, 83% which, while quite high, certainly is not high enough to ensure long repeated sequences of 412 triples. Looking at the length distribution, it turns out that the length 91 is far and away the most likely length in this run (an order of magnitude more likely than any other length in P_L). 91 instructions is obviously *far* more than necessary to solve this problem, so a successful 91 instruction program based on the 412 pattern is going to have to "get lucky" and have some instructions re-write `R1` to x towards the end, leaving just the right number of 412 triples at the end. This, however, requires that there is some reasonable chance of "escaping" runs of 412, which presumably accounts for the lower than expected probability for this key triplet. If this run had settled on a shorter length, one might expect the probability of generating the 412 sequence to be higher.

Tests with functions with more complex structure, e.g., the polynomials of the form $x^d+2x^{d-1}+3x^{d-2}+\cdots+d\,x$ (not reported due to space limitations), show that N-gram GP is capable of learning correlations that allow it to construct important sequences that are considerably longer than the triplets actually tracked in the distribution matrices. For example, in one case, N-gram GP solve a problem by learning a sequence of 9 instructions and setting triplets probabilities so that this sequence was generated with roughly 63% probability, which is over 500,000 times more likely than choosing it at random from a uniform distribution of sequences of seven instructions. N-gram GP is thus clearly capable of capturing and using long distance correlations despite only tracking the distribution of 3-grams.

4.2 Lawnmower

In contrast to the polynomial regression problems discussed above, the lawnmower problem requires much less precision in the sequence of instructions, and consequently the distribution matrices do not converge as strongly on a specific sequence.

One representative run, for example, is capable of generating a whole variety of sequences of instructions, with much less obvious patterning, including sequences such as 0000000001002..., 0000000020202..., and 2001122020020.... Still, while there are not clear sequences of instructions, there are definite trends in the distribution of instructions. This run has a high probability (74%) of starting with a `Mow` instruction, and given an initial `Mow` instruction, again a high probability (75%) of following that

with a second `Mow` instruction. In general `Mow` instructions are very likely throughout, as we would expect, while other combinations are quite uncommon. A `Mow-Right` pair has effectively no chance of being followed by a `Left`, which seems reasonable as a `Right-Left` sequence is effectively a NO-OP; in fact a `Mow-Right` pair is almost always followed by another `Mow`.

As an indication of the trends in the evolved distribution matrices, there are five initial sequences of length 8 that have a cumulative probability of at least 0.0001 of being generated: 00000000, 00000200, 00002000, 00020000 and 00200000. `Mow` instructions obviously dominate, with at most one other instruction in each case (which is in fact always a `Right`). So, in the lawnmower problem, instead of learning specific sequences of instructions to mow the lawn, N-gram GP develops stochastic space filling strategies.

5 Conclusions

We presented N-gram GP, an EDA for the evolution of linear computer programs. The algorithm learns and samples the joint probability of 3-grams of instructions at the same time as it is learning and sampling a program length distribution.

This work has several interesting features. Firstly, while several authors have extended EDAs to evolve computer programs, virtually all have done so for a tree representation. Here, for the first time, we explore the application of EDAs to a linear GP. The second distinctive feature of this work is that we explictly represent the program length distribution to be used during the search. With tree-based representations this is not used, since the primitive set always includes terminals, which, if drawn with sufficiently high frequency can terminate the construction of programs. A disadvantage of relying on terminal selection is that the probability of drawing terminals is totally under the control of evolution. Thus the user has no control over the size of the evolved programs, which, if unchecked, can easily become excessive. So, often hard limits on program depth or length must be artificially enforced, producing an undesirable bias. Here, by explicitly modelling the size distribution we instead have a natural way of limiting the search to programs of manageable size, without introducing any undesired bias. Finally, previous work has tended to use different probability distributions for different positions (or loci) in a tree, thereby expanding significantly the size of the parameter space in which the model lives. This is not a problem *per se*, but one must keep in mind that the more parameters a model has the more information must be collected from the search space to properly set those parameters, while still avoiding over-fitting. In N-gram GP we borrow from the field of natural language processing, using n-grams to represent regularities in the language necessary to solve a problem. This means that we use the same distribution for all loci in a program (except the first two, of course). This leads to a much smaller model space, where models can thus reliably be identified with less sampling, and with a higher degree of regularity in the evolved solutions.[2]

In our tests with two problem classes the N-gram GP system has been a very effective solver. Furthermore, its scalability has been significantly better than the simple

[2] Regular solutions to a problem are normally more compact and easier to interpret. However, whether or not highly regular solutions exist for a particular problem, or are easier to find than irregular ones, depends on the problem. If there is regularity, N-gram GP can exploit it.

hill-climber and linear GP, leading it to routinely solve problems of a difficulty which is way beyond what can be tackled by the other two algorithms tested.

References

1. Abbass, H., Hoai, N., McKay, R.: AntTAG: A new method to compose computer programs using colonies of ants. In: IEEE Congress on Evolutionary Computation (2002)
2. Baluja, S., Caruana, R.: Removing the genetics from the standard genetic algorithm. In: Prieditis, A., Russell, S. (eds.) Machine Learning: Proceedings of the Twelfth International Conference, pp. 38–46. Morgan Kaufmann Publishers, San Francisco (1995)
3. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge (1994)
4. Larrañaga, P., Lozano, J.A.: Estimation of Distribution Algorithms, A New Tool for Evolutionary Computation. Kluwer Academic Publishers, Dordrecht (2002)
5. Manning, C., Schütze, H.: Foundations of statistical natural language processing. MIT Press, Cambridge (1999)
6. Mühlenbein, H., Mahnig, T.: Convergence theory and application of the factorized distribution algorithm. Journal of Computing and Information Technology 7(1), 19–32 (1999)
7. Nordin, P.: A compiling genetic programming system that directly manipulates the machine code. In: Kinnear Jr, K.E. (ed.) K, ch. 14, pp. 311–331. MIT Press, Cambridge (1994)
8. Poli, R., McPhee, N.F.: A linear estimation-of-distribution GP system. Tech. Report CES-479, Dept. of Computing and Electronic Systems, University of Essex (January 2008)
9. Rabiner, L.: A tutorial on hidden Markov models and selected applications in speech recognition. Proceedings of the IEEE 77(2), 257–286 (1989)
10. Ratle, A., Sebag, M.: Avoiding the bloat with probabilistic grammar-guided genetic programming. In: Collet, P., Fonlupt, C., Hao, J.-K., Lutton, E., Schoenauer, M. (eds.) EA 2001. LNCS, vol. 2310, pp. 255–266. Springer, Heidelberg (2002)
11. Salustowicz, R.P., Schmidhuber, J.: Probabilistic incremental program evolution. Evolutionary Computation 5(2), 123–141 (1997)
12. Sastry, K., Goldberg, D.E.: Probabilistic model building and competent genetic programming. In: Riolo, R.L., Worzel, B. (eds.) Genetic Programming Theory and Practise, ch. 13, pp. 205–220. Kluwer, Dordrecht (2003)
13. Shan, Y., McKay, R.I., Abbass, H.A., Essam, D.: Program evolution with explicit learning: a new framework for program automatic synthesis. In: Sarker, R., Reynolds, R., Abbass, H., Tan, K.C., McKay, B., Essam, D., Gedeon, T. (eds.) Proceedings of the 2003 Congress on Evolutionary Computation CEC 2003, Canberra, December 2003, pp. 1639–1646. IEEE Press, Los Alamitos (2003)
14. Shan, Y., McKay, R.I., Essam, D., Abbass, H.A.: A survey of probabilistic model building genetic programming. In: Pelikan, M., Sastry, K., Cantu-Paz, E. (eds.) Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications, Springer, Heidelberg (2006)
15. Suen, C.Y.: *n*-gram statistics for natural language understanding and text processing. IEEE Transactions on Pattern Analysis and Machine Intelligence 1(2), 164–172 (1979)
16. Yanai, K., Iba, H.: Estimation of distribution programming based on bayesian network. In: Sarker, R., Reynolds, R., Abbass, H., Tan, K.C., McKay, B., Essam, D., Gedeon, T. (eds.) Proceedings of the 2003 Congress on Evolutionary Computation CEC 2003, pp. 1618–1625. IEEE Press, Los Alamitos (2003)
17. Yanai, K., Iba, H.: Program evolution by integrating EDP and GP. In: Deb, K., et al. (eds.) GECCO 2004. LNCS, vol. 3102, pp. 774–785. Springer, Heidelberg (2004)

Feature Discovery in Reinforcement Learning Using Genetic Programming

Sertan Girgin[1] and Philippe Preux[1,2]

[1] Team-Project SequeL, INRIA Futurs Lille
[2] LIFL (UMR CNRS), Université de Lille
{sertan.girgin,philippe.preux}@inria.fr

Abstract. The goal of reinforcement learning is to find a policy that maximizes the expected reward accumulated by an agent over time based on its interactions with the environment; to this end, a function of the state of the agent has to be learned. It is often the case that states are better characterized by a set of features. However, finding a "good" set of features is generally a tedious task which requires a good domain knowledge. In this paper, we propose a genetic programming based approach for feature discovery in reinforcement learning. A population of individuals, each representing a set of features, is evolved, and individuals are evaluated by their average performance on short reinforcement learning trials. The results of experiments conducted on several benchmark problems demonstrate that the resulting features allow the agent to learn better policies in a reduced amount of episodes.

1 Introduction

Reinforcement learning (RL) is the problem faced by an agent that is situated in an environment and must learn a particular behavior through repeated trial-and-error interactions with it [1]; at each time step, the agent observes the state of the environment, chooses its action based on these observations and in return receives some kind of "reward", in other words a *reinforcement signal*, from the environment as feedback. Usually, it is assumed that the decision of the agent depends only on the current state but not on the previous ones, i.e. has the Markovian property. The aim of the agent is to find a policy, a way of choosing actions, that maximizes its overall gain. Here, the gain is defined as a function of rewards, such as the (discounted) sum or average over a time period. Unlike supervised learning problem, in RL correct input/output pairs, i.e. optimal action at a given situation, are not presented to the agent, nor sub-optimal actions explicitly corrected. One key aspect of RL is that the rewards can be *delayed* in the sense that immediate rewards received by the agent may not be reflecting the true values of the chosen actions. For example, in the game of chess a move which causes your opponent to capture a piece of yours can be regarded as a "bad" move. However, a series of such moves on purpose may be essential to win the game and consequently receive a higher reward in the future.

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 218–229, 2008.
© Springer-Verlag Berlin Heidelberg 2008

There are two main approaches for solving RL problems. In the first approach, the agent maintains a function $V^{\pi}(s)$, called *value function*, that estimates the expected return when starting in state s and following policy π thereafter, and tries to converge to the value function of the optimal policy. The policy is inferred from the value function. Alternatively, in *direct policy search* approaches, the policy is represented as a parameterized function from states to actions and an optimal policy is searched directly in the space of such functions. There also exist methods that combine both approaches. Note that, in any case, the functions that we are learning (either value function, policy, or both) are naturally functions of the state (observation) variables. However, in a given problem (i) all these variables may not be relevant, which leads to *feature selection* problem, i.e. selecting a subset of useful state variables, or worse (ii) in their raw form they may be inadequate for successful and/or efficient learning and it may be essential to use some kind of *feature discovery*.

The most obvious situation where the second case emerges is when the number of states is large, or infinite, and each state variable reflects limited and local information about the problem. Let us consider the popular game of Tetris. In Tetris, traditionally each state variable corresponds to the binary (occupied/empty) status of a particular cell of the grid. Not only the number of possible states increases exponentially with respect to the size of the grid, but also each state variable tells very little about the overall situation. A human player (most successful computer players as well) instead takes into consideration more informative features that are computable from the state variables, such as the height of each column or the number of holes in the occupied regions of the grid, and decides on his actions accordingly. Similar reductions are also quite common in other domains, such as image processing applications where instead of the raw bitmap various high level features derived from it are fed into learning algorithms. On the other end of the spectrum, in some cases, additional features can be useful to improve the performance of learning. An example of this situation is presented in Fig. 1 for the classical cart-pole balancing problem. By adding sine and cosine of the pole's angle as new features to existing state variables, optimal policy can be attained much faster. A related question is, of course, given a problem what these features are and how to find them. Note that feature discovery, which will also be our main objective, is a more general problem and includes feature selection as a special case.

Usually, the set of features that are to be used instead of or together with state variables are defined by the user based on extensive domain knowledge. They can either be fixed, or one can start from an initial subset of possible features and iteratively introduce remaining features based on the performance of "the current set", this is the *feature iteration approach* [2]. However, as the complexity of the problem increases it also gets progressively more difficult to come up with a good set of features. Therefore, given a problem, it is highly desirable to find such features automatically solely based on the observations of the agent. In this paper, we report using a Genetic Programming (GP) [3] based approach for that purpose. Our aim is to find functions of state variables (and

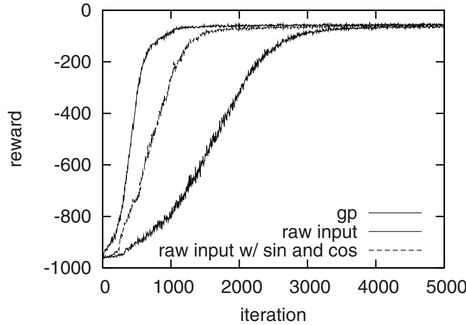


Fig. 1. In the cart-pole problem, the objective is to hold a pole, which is attached to a cart moving along a track, in upright position by applying force to the cart. The state variables are the pole's angle and angular velocity and the cart's position and velocity. Learning performance of policy gradient algorithm with Rprop update [4] when sine and cosine of the angle of the pole are added as new features, and using the features found by GP.

possibly other basis functions) that, when used as input, result in more efficient learning. Without any prior knowledge of the form of the functions, GP arises as a natural candidate for search and optimization within this context. Compared to other similar methods, such as neuro-evolutionary algorithms, it allows the user to easily incorporate domain knowledge into the search by specifying the set of program primitives. Furthermore, due to their particular representation, the resulting functions have the highly desirable property of being interpretable by humans and therefore can be further refined manually if necessary.

The rest of the paper is organized as follows: In Sect. 2, we review related work on feature discovery in RL and GP based approaches. Section 3 describes our method in detail. Section 4 presents empirical evaluations of our approach on some benchmark problems. We conclude in Sect. 5 with a discussion of results and future work.

2 Related Work

Feature discovery is essentially an information transformation problem; the input data is converted into another form that "better" describes the underlying concept and relationships, and "easier" to process by the agent. As such, it can be applied as a preprocessing step to a wide range of problems and it has attracted attention from the data-mining community.

In [5], Krawiec studies the change of representation of input data for machine learners and genetic programming based construction of features within the scope of classification. Each individual encodes a *fixed* number of new feature definitions expressed as S-expressions. In order to determine the fitness of an individual, first a new data set is generated by computing feature values for all training examples and then a classifier (decision tree learner) is trained on this

data set. The resulting average accuracy of classification becomes the evaluation of the individual. He also proposes an extended method in which each feature of an individual is assigned a utility that measures how valuable that feature is and the most valuable features are not involved in evolutionary search process. The idea behind this extension is to protect valuable features from possible harmful modifications so that the probability of accidentally abandoning promising search directions would be reduced. It is possible to view this extension as an elitist scheme at the level of an individual. While Krawiec's approach has some similarities with our approach presented in this paper, they differ in their focus of attention and furthermore we consider the case in which the number of features is not fixed but also determined by GP.

Smith and Bull [6] have also used GP for feature construction in classification. However, they follow a layered approach: in the first stage, a fixed number of new features (equal to the number of attributes in the data set subject to a minimum 7) is generated as in Krawiec (again using a decision tree classifier and without the protection extension), and then a genetic algorithm is used to select the most predictive ones from the union of new features and the original ones by trying different combinations. Although their method can automatically determine the number of features after the second stage, in problems with a large number of attributes the first stage is likely to suffer as it will try to find a large number of features as well (which consequently affects the second stage).

In RL, Sanner [7] recently introduced a technique for online feature discovery in relational reinforcement learning, in which the value function is represented as a ground relational naive Bayes net and structure learning is focused on frequently visited portions of the state space. The features are relations built from the problem attributes and the method uses a variant of the *Apriori* algorithm to identify features that co-occur with a high frequency and creates a new joint feature as necessary.

The more restricted feature selection problem can easily be formulated using a fixed-length binary encoding. It has been extensively studied within the soft computing community, and in particular, various genetic algorithm (GA) based methods have been proposed. One of the earliest works is by Siedlecki and Sklansky in which GA is used to find the smallest subset of features such that the performance of a classifier meets the specified criterion [8]. We refer the interested reader to [9] and [10] for reviews of related work.

3 Feature Discovery in RL Using GP

When GP is being applied to a particular problem, there are three main issues that need to be addressed: (i) structure and building blocks (i.e. primitive functions and terminals) of the individuals, (ii) set of genetic operators, and (iii) fitness function. In our case, due to the fact that we are interested in identifying useful features for a given RL problem, each individual must essentially be a program that generates a set of features based on the state variables. Consequently, state variables are the *independent variables* of the problem and are included in

the set of terminals together with (ephemeral) constants and possible problem specific zero argument functions. An individual consists of a list of S-expressions, called *feature-functions*; each S-expression corresponds to a unique feature represented as a function of various arithmetic and logical operators and terminals in parenthesized prefix notation. This particular representation of an S-expression lends itself naturally to a tree structure. Given the values of state variables, the values of features can be calculated by traversing and evaluating the corresponding S-expressions. In our implementation, instead of directly using the tree forms, we linearized each S-expression in prefix-order and then concatenated them together to obtain the final encoding of the individual. This compact form helps to reduce memory requirements and also simplifies operations. As we will describe later, each individual is dynamically compiled into executable binary form for evaluation and consequently this encoding is accessed/modified only when genetic operators are applied to the individual.

Note that the number of useful features is not known a priori (we are indeed searching for them) and has to be determined. Instead of fixing this number to an arbitrary value, we allowed the individuals to accommodate varying number of feature functions (S-expressions) within a range, typically less than a multiple of the number of raw state variables, and let the evolutionary mechanism search for an optimal value. To facilitate the search, in addition to regular genetic operators presented in Table 1 we also defined a single-point crossover operator over the feature function lists of two individuals. Let n and m be the number of features of two individuals selected for cross-over, and $0 < i < n$ and $0 < j < m$ be two random numbers. The first i features of the first individual are merged with the last $m - j$ features of the second individual, and the first j features of the second individual are merged with the last $n - i$ features of the first individual to generate two off-springs (Fig. 2). The generated off-springs contain a mixture of features from both parents and may have different number of features compared to them.

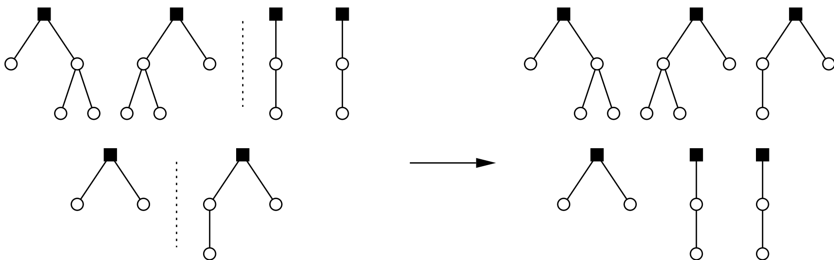


Fig. 2. Single point cross-over on feature lists of two individuals. Cross-over point is shown by vertical dashed line. The number of features represented by the off-springs differ from that of their parents.

Since our overall goal is to improve the performance of learning, the obvious choice for the fitness of an individual is the expected performance level achieved

Table 1. Genetic operators

Cross-over One of the nodes in any feature function of an individual and the whole branch under it is switched with another node from another individual in the population.

Mutation (node) One of the nodes in any feature function of an individual is substituted with another compatible one – a terminal or a zero argument function is replaced with a terminal or a zero argument function, and an n-ary operator is replaced with an n-ary operator. Note that the branch under the mutated node, if any, is not affected.

Mutation (tree) One of the nodes in any feature function of an individual and the whole branch under it is substituted with a new randomly generated sub-tree having a depth of 3 or less.

Shrinkage One of the operator nodes in any feature function of an individual is substituted with one of its children.

Feature-list cross-over See text and Fig. 2.

by the agent when the corresponding feature functions are applied on a particular RL algorithm on a particular problem. In this work, we opted for two different algorithms, namely λ policy iteration and policy gradient method with RProp update; they are described in more detail in Sect. 4. In both RL algorithms, we represented the value function and the policy as a linear combination of feature functions, hence the parameters correspond to the coefficients of each feature function. The fitness scores of individuals are calculated by taking their average performance over a small number (around 4-10) of short learning trials using the corresponding RL algorithm. In the experiments, we observed that both algorithms converge quickly towards an approximately optimal policy when the basis feature functions capture the important aspects of the complicated non-linear mapping between states and actions. We also penalized feature functions according to their size to avoid very large programs, but in practice we observed that this penalization had very little effect as feature sets consisting of simpler functions tend to perform better and receive higher scores.

Accelerating the Computations

It is well known that GP is computationally demanding. In our case, which also applies in general, there are two main bottlenecks: (i) the time required to execute the program represented by an individual, and (ii) the need to evaluate many individuals in each generation.

During the evaluation of a single individual, feature functions are called repeatedly for different values of state variables in order to calculate the corresponding feature values. If at each call, the actual tree structure of each feature function (or its linear form) is interpreted directly by traversing the S-expression, much time is spent in auxiliary operations such as following nodes, pushing/popping values onto the stack, parsing node types etc. This overhead can easily, and in fact eventually, become a bottleneck as the size of the

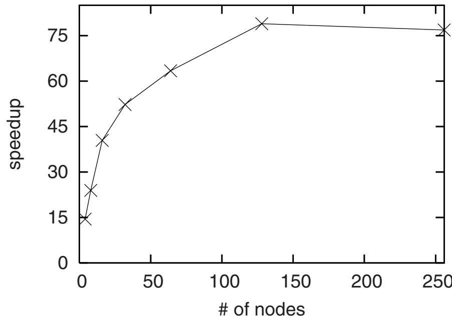


Fig. 3. Speedup for the evaluation of functions in the form of complete binary trees having depth 2-8. Each function is executed 10^6 times and the results are averaged over 10 independent runs.

individuals (i.e the number of nodes) and the number of calls (in the order of thousands or more) increase. Several approaches have been proposed to overcome this problem, such as directly manipulating machine language instructions as opposed to higher level expressions [11,12] or compiling tree representation into machine code [13]. Following the work of Fukunaga et.al. [13], we dynamically generate machine code at run-time for each feature function using GNU Lightning library, and execute the compiled code at each call. GNU Lightning is a portable, fast and easily retargetable dynamic code generation library [14]; it abstracts the user from the target CPU by defining a standardized RISC instruction set with general-purpose integer and floating point registers. As code is directly translated from a machine independent interface to that of the underlying architecture without creating intermediate data structures, the compilation process is very efficient and requires only a single pass over the tree representation or linearized form of an individual[1]. Furthermore, problem specific native operators or functions (mathematical functions etc.) can be easily called from within compiled code. Figure 3 shows the speedup of an optimized implementation of standard approach compared to the dynamically compiled code on randomly generated functions that have a complete binary tree form (i.e. contains $2^d - 1$ nodes where d is the depth of the tree). The speed-up increases with the size of the functions, reaching about 75 fold improvement which is substantial.

In GP, at each generation the individuals are evaluated independently of each other, that is the evaluation process is highly parallelizable. As such, it can be implemented efficiently on parallel computers or distributed computing systems. By taking advantage of this important property, we developed a parallel GP system using MPICH2 library [15], an implementation of the *Message Passing Interface*, and run the experiments on a Grid platform. This also had a significant impact on the total execution time.

[1] Time to compile a function of 64 nodes is around 100 microseconds on a 2.2Ghz PC.

4 Experiments

We evaluate the proposed GP based feature discovery method on three different benchmark problems: Acrobot [16], multi-segment swimmer [17] and Tetris [18]. The first two problems, Acrobot and multi-segment swimmer, are dynamical systems where the state is defined by the position and velocity of the elements of the system, and action being an acceleration which, according to Newton's law, defines the next state. These are non-linear control tasks with continuous state and action spaces; the number of state variables are respectively 4 and $2n+2$ where n is the number of segments of the swimmer. Despite their seemingly easiness, these two tasks are difficult to learn (to say the least, far from obvious). In Acrobot, there is only a single control variable, whereas in swimmer the agent has to decide on torques applied to each of $n-1$ joints. Although it is similar in nature to Acrobot, swimmer problem has significantly more complex state and control spaces that can be varied by changing the number of segments. As the number of segments increase the problem also becomes harder. Our third benchmark problem, the game of Tetris, has discrete state and action spaces. The state variables are the following: (i) the heights of each column, (ii) the absolute difference between the heights of consecutive columns, (iii) the maximum wall height, (iv) the number of holes in the wall (i.e. the number of empty cells that have an occupied cell above them), and (iv) the shape of the current object. We used a 12×8 grid and 7 different shapes that consist of 4 pieces, thus the number of features was 18.

For evaluating the individuals and testing the performance of the discovered features, we employed two different RL algorithms: λ policy iteration for the Tetris problem, and policy gradient method with RProp update for the Acrobot and swimmer problems. λ policy iteration is a family of algorithms introduced by Ioffe and Bertsekas which generalizes standard value iteration and policy iteration algorithms [2]. Value iteration starts with an arbitrary value function and at each step updates it using the Bellman optimality equation (with one step backup); the resulting optimal policy is greedy with respect to the value function[2]. On the other hand, policy iteration starts with an initial policy and generates a sequence of improving policies such that each policy is greedy with respect to the estimated value (calculated by policy evaluation) of its predecessor. λ policy iteration fuses both algorithms together with a parameter $\lambda \in (0,1)$ by taking a λ-adjustable step toward the value of next greedy policy in the sequence [19]. We used the approximate version of λ policy iteration as defined in Sect. 8.3 of [18]. Policy gradient method also works on the policy space, but approximates a parameterized (stochastic) policy directly. Starting from an initial policy, the policy parameters are updated by taking small steps in the direction of the gradient of its performance and under certain conditions converge to a local optima in the performance measure [20]. The gradient is usually estimated using Monte Carlo roll-outs. With RProp update, instead of directly relying on the magnitude of the gradient for the updates (which may

[2] For example, that selects in each state the action with highest estimated value.

lead to slow convergence or oscillations depending on the learning rate), each parameter is updated in the direction of the corresponding partial derivative with an individual time-varying value. The update values are determined using an adaptive process that depends on the change in the sign of the partial derivatives.

In the experiments, a population consisting of 100 individuals evolved for 50 generations. We set crossover probability to 0.7 (with a ratio of 1/6 for the feature-list crossover), and the remaining 0.3 is distributed among mutation and shrinkage operators. Node mutation is given two times higher probability than the others. There is a certain level of elitism, 10% of best performing individuals of each generation are directly transferred to the next generation. The set of operators is $\{+, -, *, /, \sin, \cos \sqrt{\cdot}\}$ for the Acrobot and swimmer problems and $\{+, -, *, /, \min, \max, \|-\|_1\}$ for the Tetris problem where $\|-\|_1$ denotes the absolute difference between two values. The terminals consist of the set of original state variables as given above and $\{1, 2, e\}$ where e denotes an ephemeral random constant $\in [0, 1]$. In the policy gradient method (Acrobot and swimmer problems), an optimal baseline is calculated to minimize the variance of the gradient estimate, and the policy is updated every 10 episodes. We tested with two different sets of parameters: ($\triangle_{min} = 0.01, \triangle_{ini} = 0.1, \triangle_{max} = 0.5$) and ($\triangle_{min} = 0.002, \triangle_{ini} = 0.02, \triangle_{max} = 0.1$). In λ policy iteration (Tetris), we run 30 iterations and sampled 100 trajectories per iteration using the greedy policy at that iteration. λ is taken as 0.6. The results presented here are obtained using a single GP run for each problem. The discovered features are then tested on the same RL algorithms but with a longer training period (50000 iterations for policy gradient, and 30 iterations for λ policy iteration) to verify how well they perform. We averaged over 20 such test trainings.

Figure 4a and Fig. 5[a,c] show the testing results for the Acrobot and multiple-segment swimmer problems, respectively. In both cases, features found by GP show an improvement over original features and allow the agent to learn policies with larger return. The improvement is more evident in swimmer problem, where the agents utilizing the discovered features can learn policies that perform on average 50% better. Since candidate feature-functions are evaluated based on their average performances on short learning trials, learning speed is also faster in the initial stages as expected. The fitness values of best individuals and average fitness of the population in each generation during the feature discovery process indicate that the evolutionary search drives towards better solutions and further improvement may be possible with longer GP runs especially in swimmer problem (Fig. 4b and Fig. 5[b,d]). Note that, the learning curves are not strictly increasing due to stochasticity in the simulations. We obtained inferior results with smaller population sizes (Fig. 5b). Although we used a different RL algorithm, the results for Tetris are also similar to those of Acrobot and swimmer, and show considerable improvement in terms of the performance of the resulting policies (Fig. 6).

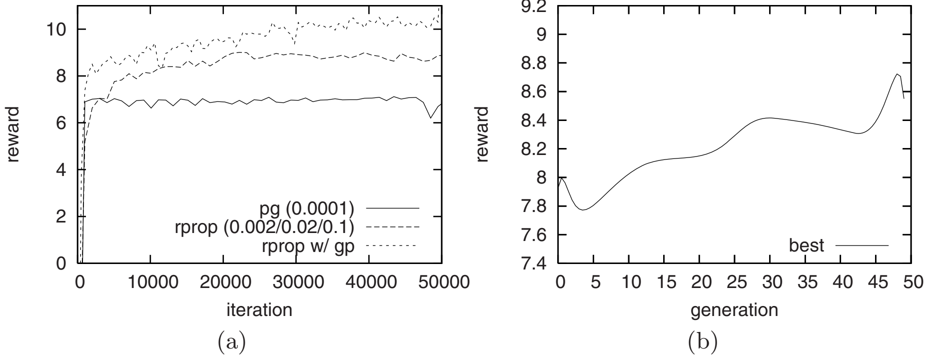


Fig. 4. Results for Acrobot. (a) Performance of discovered features, and (b) fitness values of best individuals in each generation.

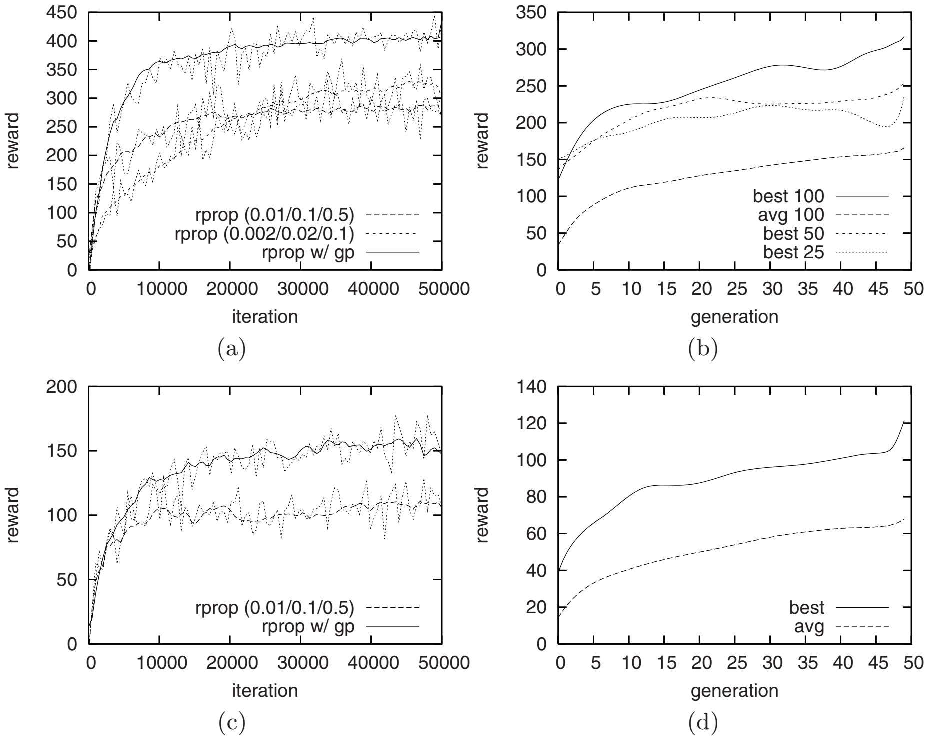


Fig. 5. Results for 3 (Fig. a and b) and 5 (Fig. c and d) segment swimmers. (a, c) Performance of discovered features, and (b, d) fitness values of best individuals and average fitness of the population in each generation. Figure b also shows the fitness values of best individuals for the population sizes of 25 and 50.

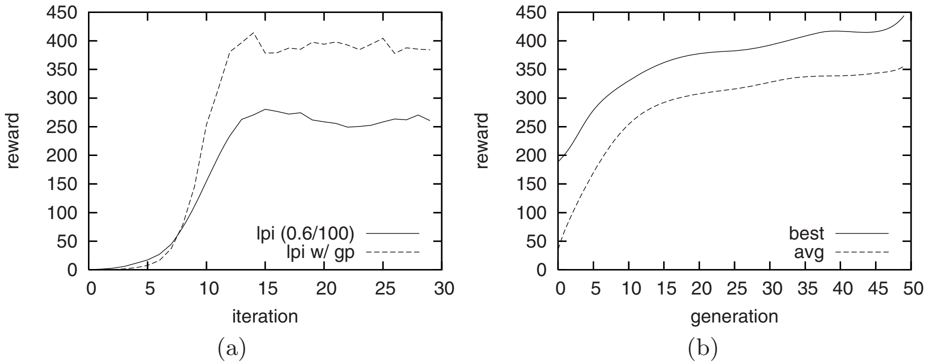


Fig. 6. Results for Tetris. (a) Performance of discovered features, and (b) fitness values of best individuals and average fitness of the population in each generation.

5 Conclusion

In this paper, we explored a novel genetic programming based approach for discovering useful features in reinforcement learning problems. Empirical results show that evolutionary search is effective in generating functions of state variables that when fed into RL algorithms allow the agent to learn better policies. As supported by previous results in classification tasks, the approach may also be applicable in supervised settings by changing the learning algorithm used for evaluating the individuals. However, care must be taken to choose algorithms that converge quickly when supplied with a "good" state representation.

One important point of the proposed method is that it allows the user to guide the search and if needed incorporate domain knowledge simply by specifying the set of program primitives (i.e. ingredients of the feature functions). Furthermore, resulting feature functions are readable by humans (and not hard to comprehend) which makes it possible to fine-tune and also transfer knowledge to (feature extraction process of) similar problems. This can be done either manually, or by converting them into meta functions, as in *automatically defined functions* [21], leading to a hierarchical decomposition. We pursue future research in this direction.

Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see https://www.grid5000.fr).

References

1. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998) (A Bradford Book)
2. Bertsekas, D., Ioffe, S.: Temporal differences-based policy iteration and applications in neuro-dynamic programming. Technical Report LIDS-P-2349, MIT (1996)
3. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
4. Riedmiller, M., Peters, J., Schaal, S.: Evaluation of policy gradient methods and variants on the cart-pole benchmark, pp. 254–261 (2007)
5. Krawiec, K.: Genetic programming-based construction of features for machine learning and knowledge discovery tasks. Genetic Programming and Evolvable Machines 3(4), 329–343 (2002)
6. Smith, M.G., Bull, L.: Genetic programming with a genetic algorithm for feature construction and selection. Genetic Programming and Evolvable Machines 6(3), 265–281 (2005)
7. Sanner, S.: Online feature discovery in relational reinforcement learning. In: Open Problems in Statistical Relational Learning Workshop (SRL-2006) (2006)
8. Siedlecki, W., Sklansky, J.: A note on genetic algorithms for large-scale feature selection. Pattern Recogn. Lett. 10(5), 335–347 (1989)
9. Martin-Bautista, M.J., Vila, M.A.: A survey of genetic feature selection in mining issues. In: Proceedings of the 1999 Congress on Evolutionary Computation CEC 1999, vol. 2, p. 1321 (1999)
10. Hussein, F.: Genetic algorithms for feature selection and weighting, a review and study. In: Proceedings of the Sixth International Conference on Document Analysis and Recognition, Washington, DC, USA, p. 1240. IEEE Computer Society, Los Alamitos (2001)
11. Nordin, P.: A compiling genetic programming system that directly manipulates the machine code. In: Kinnear Jr, K.E. (ed.) Advances in Genetic Programming, pp. 311–331. MIT Press, Cambridge (1994)
12. Banzhaf, W., Francone, F.D., Keller, R.E., Nordin, P.: Genetic programming: an introduction: on the automatic evolution of computer programs and its applications. Morgan Kaufmann Publishers Inc, San Francisco (1998)
13. Fukunaga, A., Stechert, A., Mutz, D.: A genome compiler for high performance genetic programming. In: Genetic Programming 1998: Proceedings of the Third Annual Conference, University of Wisconsin, Madison, Wisconsin, USA, pp. 86–94. Morgan Kaufmann, San Francisco (1998)
14. G.N.U.: Lightning (2007), http://www.gnu.org/software/lightning/
15. Laboratory, A.N.: Mpich2 (2007), http://www-unix.mcs.anl.gov/mpi/mpich2/
16. Spong, M.W.: Swing up control of the acrobot. In: ICRA, pp. 2356–2361 (1994)
17. Coulom, R.: Reinforcement Learning Using Neural Networks, with Applications to Motor Control. PhD thesis, Institut National Polytechnique de Grenoble (2002)
18. Bertsekas, D.P., Tsitsiklis, J.N.: Neuro-Dynamic Programming. Athena Scientific, Belmont, MA (1996)
19. Scherrer, B.: Performance bounds for lambda policy iteration (2007)
20. Sutton, R.S., McAllester, D.A., Singh, S.P., Mansour, Y.: Policy gradient methods for RL with function approximation. In: NIPS, pp. 1057–1063 (1999)
21. Koza, J.R.: Genetic programming II: automatic discovery of reusable programs. MIT Press, Cambridge (1994)

Hardware Accelerators for Cartesian Genetic Programming

Zdenek Vasicek and Lukas Sekanina

Faculty of Information Technology, Brno University of Technology
Božetěchova 2, 612 66 Brno, Czech Republic
vasicek@fit.vutbr.cz, sekanina@fit.vutbr.cz

Abstract. A new class of FPGA-based accelerators is presented for Cartesian Genetic Programming (CGP). The accelerators contain a genetic engine which is reused in all applications. Candidate programs (circuits) are evaluated using application-specific virtual reconfigurable circuit (VRC) and fitness unit. Two types of VRCs are proposed. The first one is devoted for symbolic regression problems over the fixed point representation. The second one is designed for evolution of logic circuits. In both cases a significant speedup of evolution (30–40 times) was obtained in comparison with a highly optimized software implementation of CGP. This speedup can be increased by creating multiple fitness units.

1 Introduction

According to John Koza, genetic programming (GP) can routinely deliver high-return human-competitive machine intelligence [1]. Its competitiveness and performance has been demonstrated in many tasks and design areas. Simultaneously, the computational power which GP needs for obtaining innovative results is enormous for most applications. GP usually spends most of time by running domain-specific simulators which evaluate candidate individuals using large training sets. In order to reduce the computational time, various methods have been employed. In general, they can be divided into four classes: (1) algorithmic – the use of smart search strategies, genetic operators and fitness evaluation strategies, (2) source code optimization for a given platform, (3) parallel GP implementations on clusters of workstations and (4) hardware accelerators. However, even with a parallel GP, the evolution is very time consuming. For example, Koza's team has utilized two clusters of workstations, 1000 x Pentium II/350 MHz processor and 70 x DEC Alpha/533 MHz processor. For 36 tasks solved using GP on the clusters, the average population size is 3,350,000 individuals, 128.7 generations are produced in average and the average time to reaching a solution is 81.9 hours [1].

This paper is focused on the acceleration of GP using a suitable digital hardware. For genetic algorithms, FPGA (Field Programmable Gate Arrays) based implementations have been created for a long time [2, 3]. As the fitness evaluation of a candidate program is the most time consuming part of GP, hardware

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 230–241, 2008.
© Springer-Verlag Berlin Heidelberg 2008

acceleration should primarily be devoted to the fitness calculation. A straighforward implementation involves multiple fitness calculation units which work concurrently. Another key issue in hardware is whether the particular problem requires the floating-point (FP) operations or fixed-point (FX) operations. The fixed-point arithmetic circuits or even logic circuits can be accelerated in a much easier way than floating-point operations on a commonly accessible hardware such as FPGA. Martin implemented a complete linear genetic programming system in an FPGA. It operates with FX expressions encoded as linear programs. Depending on the number of hardware fitness evaluation units, he reported the speedup 18 (for 2 fitness units) - 419 (64 fitness units) for the even 6-parity problem and 13 (2 fitness units) - 107 (32 fitness units) for the artificial ant problem in comparison with the PowePC processor running at 200 MHz [4].

Recently, Graphics Processing Units (GPUs) that are available in common desktop computers have been used to parallelize the fitness evaluation (also for the FP domain) [5, 6, 7]. The CPU converts arrays of test cases to textures on the GPU and loads a shader program into the shader processors. According to a GP expression, a shader program is created. The program is then executed, and the resulting texture is converted back in to an array. The fitness is determined from this output array [6]. Chitty [7] reports the speedup 0,4 – 30 depending on target problem (two symbolic regressions, Iris classification and 8-input multiplexer tested) for NVidia GeForce 6400 GO graphic card in comparison with a 1.7 GHz Pentium 4 processor. Harding and Banzhaf have shown how the speedup of candidate individual evaluation depends on the expression length for various problems. With the growing expression length and growing number of test cases, GPU becomes more effective than CPU. The maximum speedup is approx. 1000 for Boolean expressions and 14 for a protein classification problem. Note that these results only show the number of times faster evaluating evolved GP expressions is on the GPU (NVidia GeForce 7300 GO) compared to CPU implementation (Intel Centrino T2400 running at 1.83 GHz). I.e., the speedup of evolution was not reported. Unfortunately, for training sets of a common size, the overhead of transferring data to the GPU and for constructing shaders leads to a worse performance than CPU.

In the recent years, human-competitive results were obtained using Cartesian Genetic Programming (CGP) [8]. CGP is a sort of genetic programming which represents candidate programs as graphs consisting of an array of programmable nodes. This representation is natural for hardware implementation. In this paper, we propose an approach to building CGP accelerators in an FPGA. The accelerator consists of genetic unit, fitness unit and the so-called virtual reconfigurable circuit (VRC) which is utilized to evaluate candidate programs. We will show that even if only a single fitness unit operating at 100 MHz is utilized, the evolution is 30–40 times faster than a highly optimized software implementation running at a GHz processor. This approach is well suited especially for integer-level symbolic regression problems and evolution of logic expressions. The implementation utilizes a commercial off-the-shelf FPGA Virtex II Pro which contains sufficient logic resources and on-chip PowerPC processors. As genetic operations

are implemented in the PowerPC processor, the designer can define a new target problem and change various parameters of CGP very quickly.

The proposed solution was originally intended for evolution of image filters. In this particular problem, human competitive results were obtained because "the result (i.e. image filters presented in [9]) is publishable in its own right as a new scientific result – independent of the fact that the result was mechanically created" (criterion D from [10]). The goal of this paper is to demonstrate that the method can be extended to be considered as a general CGP accelerator for those problems which utilize FX operators or logic operators. The VRCs for typical target domains will be presented together with an analysis of the impact of their parameters on the performance. In particular, we will investigate the effect of setting the level of interconnectivity (the L-back parameter of CGP). A new hardware approach will be presented which allows optimizing not only for function but also for the size of a candidate program (not reported so far in literature). In addition to the use of multiple fitness units and pipelining, we will also introduce a new parallel approach to the evaluation of candidate programs. The accelerators will be evaluated using benchmark problems commonly used in this area.

2 Cartesian Genetic Programming

In CGP, a candidate program is modeled as an array of u (columns) $\times$ v (rows) of programmable elements (gates). The number of inputs, n_i, and outputs, n_o, is fixed. Feedback is not allowed. Each node input can be connected either to the output of a node placed in the previous L columns or to some of program inputs. The L-back 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 node is programmed to perform one of functions defined in the set Γ (n_f denotes $|\Gamma|$). As Figure 1 shows, while the size of chromosome is fixed, the size of phenotype is variable (i.e. some nodes are not used). Every individual is encoded using $u \times v \times 3 + n_o$ integers.

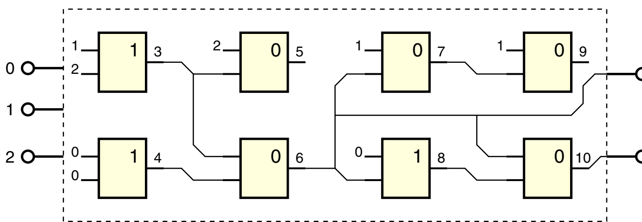


Fig. 1. An example of a candidate program. CGP parameters are as follows: $L = 3$, $u = 4$, $v = 2$, $\Gamma = \{\text{AND (0), OR (1)}\}$. Nodes 5 and 9 are not utilized. Chromosome: 1,2,1, 0,0,1, 2,3,0, 3,4,0 1,6,0, 0,6,1, 1,7,0, 6,8,0, 6, 10. The last two integers indicate the outputs of the program.

CGP operates with the population of λ individuals (typically, $\lambda = 5 - 20$). The initial population is randomly generated. Every new population consists of the best individual and its mutants. In case when 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 fitness function usually takes one of two forms. For the symbolic regression problems, a training set is used. The goal is to minimize the difference between the output of a candidate program and required output. For evolution of logic circuits, all possible input combinations are applied at the candidate circuit inputs, the outputs are collected and the goal to minimize the difference between obtained truth table and required truth table. In case when the evolution has found a solution which produces correct outputs for all possible input combinations, other parameters, such the number of components or delay are getting to minimize. The evolution is stopped when the best fitness value stagnates or the maximum number of generations is exhausted.

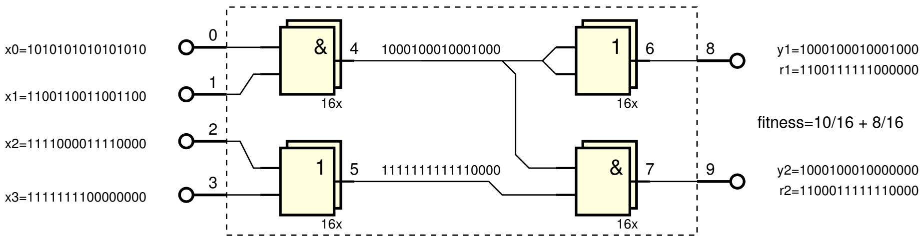


Fig. 2. Parallel simulation of a combinational circuit. Values $y1$ and $y2$ are the results of simulation, $r1$ and $r2$ are the required outputs.

Software implementations of CGP, which are intended for evolution of logic circuits, strongly benefit from the so-called *parallel simulation*. In a circuit simulator working at the gate level, a single gate is usually modeled using a logic function. The idea of parallel simulation is to utilize bitwise operators operating on multiple bits in a high-level language (such as C) to perform more than one evaluation of a gate in a single step. Therefore, when a combinational circuit under simulation has four inputs and it is possible to concurrently perform bitwise operations over $2^4 = 16$ bits in the simulator then this circuit can completely be simulated by applying a single 16-bit test vector at each input (see encoding in Fig. 2). In contrast, when it is impossible then sixteen four-bit test vectors must be applied sequentially. Practically, current processors allow us to operate with 64 bit operands, i.e. it is possible to evaluate the truth table of a six-input circuit by applying a single 64-bit test vector at each input. Therefore, the obtained speedup is 64 against the sequential simulation. In case that a circuit has more than 6 inputs then the speedup is constant, i.e. 64. This technique can be also utilized in hardware. However, it is mainly useful for gate-level evolution.

In case of function-level evolution, for example, over b-bit operators (such as addition, subtraction, maximum etc.) the speedup is only c/b, where c is the number of bits of the operators implemented in hardware.

3 Accelerators for CGP

The basic idea of proposed accelerator is that a given instance of CGP (i.e. a reconfigurable graph consisting of $u \times v$ programmable nodes) is implemented as a reconfigurable circuit on the FPGA. Its configuration is defined using a bitstream which is stored in a configuration register implemented also in the FPGA. This concept is called the virtual reconfigurable circuit [11]. In order to evaluate a candidate chromosome, a controller has to store the chromosome into the configuration register of VRC and activate the fitness unit (FU). FU generates the input vectors for VRC, reads the output vectors from VRC and compares them with required output vectors. The fitness value is sent to the on-chip PowerPC processor where new candidate chromosomes are created. This architecture was introduced in [12].

3.1 Architecture Overview

The proposed CGP accelerator is completely implemented in a single FPGA and consists of Genetic unit (GU), Processor and Memory Interface (PMI), Fitness Unit (FU), VRC and a Control Unit (CU) – a communication interface to a common PC (see Fig. 3). The PC is used just to define parameters of CGP and target data (the truth table or training set). External SRAM memories are used to store large training sets (e.g. training images for designing of image filters), while on-chip BlockRAM (BRAM) memories are used to store small training sets.

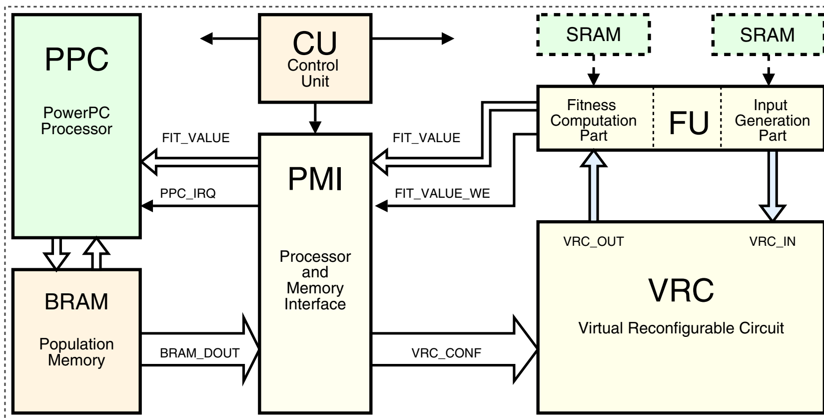


Fig. 3. Generic architecture of CGP accelerators in the FPGA Virtex 2 Pro

All components (except the VRC) are connected to the internal bus called LocalBus which provides an effective communication interface between FPGA and PCI bus. In order to maximize the overall performance, the CU plays the role of master, controls the entire system and provides an interface to the host PC. The PowerPC generates a new candidate individual when a requirement is specified. The instruction memory of the PowerPC is implemented using BRAMs. However, our search algorithm can completely be stored in an instruction cache.

The population of candidate configurations is also stored in on-chip BRAM memories. The population memory is divided into banks; each of them contains a single configuration bitstream of VRC. An additional bit (associated with every bank) determines data validity; only valid configurations can be evaluated. In order to overlap the evaluation of a candidate configuration with generating a new candidate configuration, at least two memory banks have to be utilized. While a circuit is evaluated, a new candidate configuration is generated. The new configuration is used immediately after completing the evaluation of the previous one.

The PMI component consists of two subcomponents working concurrently. The first subcomponent, controlled by the CU, reconfigures the VRC using configurations stored in the population memory. The second subcomponent is responsible for sending the fitness value to the PowerPC processor. As soon as the fitness value is valid, it is sent (together with some additional data, such as the size of phenotype) to the PowerPC. An interrupt (IRQ) is generated to activate a service routine of the PowerPC. In this routine, a new candidate configuration is generated for the given bank. The PowerPC processor acknowledges the interrupt (IRQACK) and sets up the validity bit. This process is controlled by the FU. The PMI component also provides an interface to the population memory via LocalBus.

The proposed system allows the use of various search algorithms [12]. These algorithms utilize a population of candidate solutions and a single genetic operator — mutation, which inverts k bits of the chromosome (i.e. of the configuration). No crossover operator is used. An analysis of various mutation operators and pseudorandom number generators was presented in [12, 13].

3.2 VRC for Symbolic Regression Problems

Proposed CGP accelerators mainly differ in the VRC organization and fitness unit. Fig. 4 shows the VRC implemented for the image filter design problem, which is a kind of a symbolic regression problem over the FX representation [12]. Every candidate program (image filter) is considered as a digital circuit of nine 8-bit inputs and a single 8-bit output.

The VRC consists of 2-input Configurable Logic Blocks (CFBs), denoted as E_i, placed in a grid of 8 columns and 4 rows. Any input of each CFB may be connected either to a primary circuit input or to the output of a CFB, which is placed anywhere in the preceding column. Any CFB can be programmed to implement one of 16 function from Γ, where Γ includes addition, subtraction, shift, minimum, maximum and logic functions. All these functions operate with 8-bit

operands and produce 8-bit results. The reconfiguration is performed column by column. The computation is pipelined; a column of CFBs represents a stage of the pipeline. Registers (denoted D) are inserted between the columns in order to synchronize the input pixels with CFB outputs. The configuration bitstream of VRC, which is stored in a register array *conf_reg*, consists of 384 bits. A single CFB is configured by 12 bits, 4 bits are used to select the connection of a single input, 4 bits are used to select one of the 16 functions. Evolutionary algorithm directly operates with configurations of the VRC; simply, a configuration is considered as a chromosome.

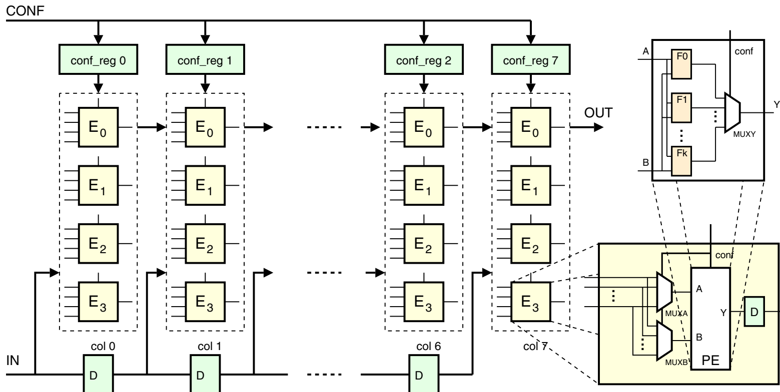


Fig. 4. VRC for symbolic regression problems

In tasks of symbolic regression, training data are stored in external SRAM memories. Fitness unit loads training data from external SRAM1 memory and forwards them to the inputs of VRC. The outputs of VRC, y_i, are compared with required outputs, r_i, (which are loaded from another external memory, SRAM2) and simultaneously stored into the third external memory, SRAM3. The FU can be considered as an extension of the VRC pipeline because in each clock cycle, a temporary fitness value is updated by a new difference, $|y_i - r_i|$. Due to pipelined reconfiguration as well as execution of VRC, the evaluation of a candidate program (circuit) requires k clock cycles, where k is the number of training vectors.

3.3 VRC for Logic Expressions

The architecture of VRC is similar to the VRC for symbolic regression. There are four main differences: PEs contain only logic functions, L-back=2 is supported, the size of phenotype can be calculated and a data parallel operation of PEs (the same as used in the software parallel simulation) is introduced. The size of data is denoted as "data width", dw, in the rest of paper. If PEs operate at dw bits then the speedup against the bit-level execution is dw-times. In order to support L-back=2, additional registers (D) have been used to store the results of stage

$i-2$ for stage i of the pipeline (see Fig. 5). The number of configuration bits for a single column is $2 * log_2(n_i + 2u) + log_2(n_f)$. In contrast to symbolic regression, the training data (truth table) is stored in BRAMs. For example, if $n_i = 16$ then 64 BRAMs are utilized. All possible input combinations are generated in the process of fitness calculation. When the size of circuit is not optimized, the maximum fitness value is $2^{n_i} n_o$.

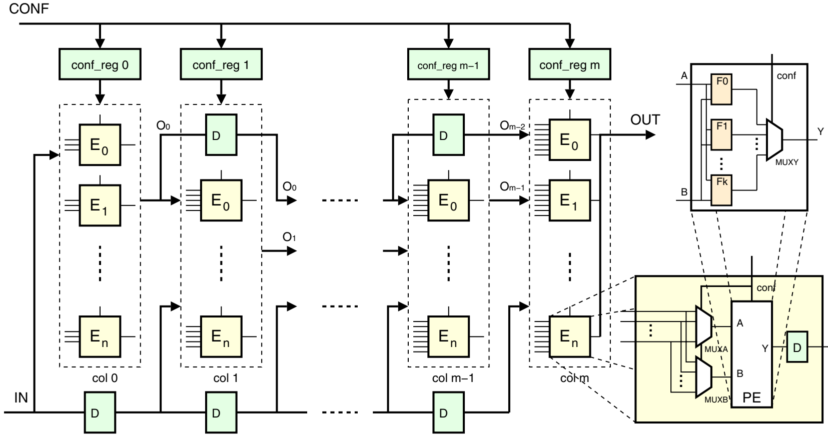


Fig. 5. VRC for evolution of digital circuits

Figure 6 explains the calculation of the size of a candidate circuit. The method assumes that a PE can implement a single wire. Once a functionally-perfect solution is found, the size is optimized. The objective is to maximize the number of PEs which operate as wires. The configuration of a single column of VRC is analyzed using comparators. The comparator returns 1 in case that a particular PE operates as a wire. These 1s are added using a tree of adders. This calculation is performed when the column of PEs is configured. It costs no extra time. The size of phenotype is stored to 8 the least significant bits of the fitness value.

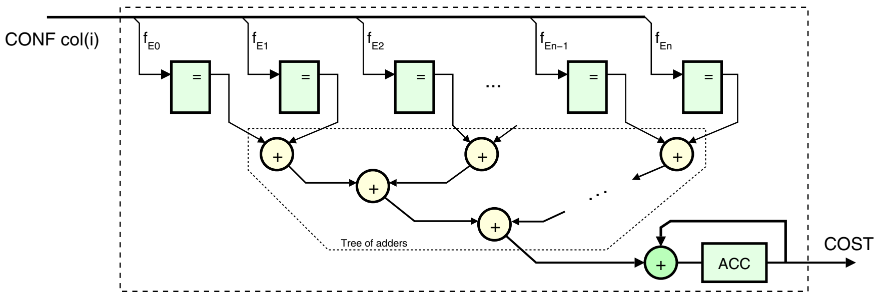


Fig. 6. Calculation of the size of a phenotype

4 Experimental Results

4.1 Evolution of Digital Circuits

Table 1 provides results of synthesis for various parameters of VRC. While the size of VRC and the number of inputs and outputs are fixed, the number of test vectors evaluated in parallel (i.e. dw) increases from 1 to 12. When no data parallel execution is used, the whole design occupies approx. 10% resources; when $dw = 12$ (i.e. 12 test vectors are evaluated in parallel by a PE) the design occupies approx. 90% resources. Using this setup we can achieve 27 times faster evaluation in comparison with a highly optimized SW implementation running at a CPU Intel Xeon 3 GHz processor (and utilizing a parallel simulation at 32 bits), even if the VRC works at 100 MHz.

Table 1. Results of synthesis for VRC with 10x10 PEs, 9 inputs, 9 outputs and 4 logic functions per PE (XC2VP50-ff1517 Xilinx FPGA). DFF is the number of flip-flops and FG is the number of function generators.

		# of vectors evaluated in parallel (dw)				
resource	**available**	**1**	**2**	**4**	**8**	**12**
BRAMs	232	14	16	20	28	36
	used	6.0%	6.9%	8.6%	12.1%	15.5%
DFFs	49788	2743	2993	3533	4709	5843
	used	5.5%	6.0%	7.1%	9.5%	11.7%
FGs	47232	4836	7813	14164	26734	41281
	used	10.2%	16.5%	30.0%	56.6%	87.4%

Table 2 contains the results of synthesis for various VRC sizes. The number of inputs, outputs, logic functions and data width are fixed. The last row shows the number of configuration bits of VRC.

Table 2. Results of synthesis for various VRCs of 9 inputs, 9 outputs, 4 logic functions and $dw = 2$ (FPGA XC2VP50-ff1517)

		VRC size			
resource	**available**	10×10	12×12	14×14	16×16
DFFs	49788	1644	2336	3634	4664
	used	3.3%	4.7%	7.3%	9.4%
FGs	47232	6242	9012	26700	32352
	used	13.2%	19.1%	56.5%	68.5%
# of conf. bits		1200	2016	2744	3584

In order to investigate the impact of the L-back parameter, we created two VRCs with $L = 1$ and $L = 2$. Proposed implementations were evaluated in the task of multiplier evolution, a traditional hard benchmark problem for evolutionary circuit design. A parallel version of Hill Climbing algorithm with neighbourhood of two and population size of 8 individuals was used (see [13]).

Table 3 summarizes results of 10 independent experiments for each problem. We can see that the increasing value of L-back parameter has the positive effect on the average number of generations and the success rate. Obtained results are comparable to the best-known results [14] (where the authors allowed the maximum value of L-back parameter).

Table 3. Results for evolution of multipliers ($\Gamma = \{$wire, and, xor, $\bar{a}$ and $b\}$)

Parameters of evolution										
multiplier	**2 × 2**		**2 × 3**		**3 × 3**		**3 × 4**		**4 × 4**	
l-back	1	2	1	2	1	2	1	2	1	2
VRC	8x8	8x8	10x10	10x10	10x10	10x10	10x10	10x10	16x16	16x16
inputs	4	4	5	5	6	6	7	7	8	8
gener. (max)	10k	10k	100k	100k	1M	1M	10M	10M	20M	20M
Results										
success rate	91%	96%	92%	100%	72%	96%	18%	84%	0%	4%
gates (min)	7	7	13	13	29	24	60	45	-	125
gates (max)	19	13	20	21	45	47	67	68	-	156
gates (avg)	9	8	15	15	34	33	61	57	-	138
gener. (avg)	1.8k	1.5k	20k	13k	22k	284k	4.84M	3.84M	-	14.2M

Table 4 compares the number of evaluated candidate circuits per one second in a highly optimized SW implementation and proposed HW accelerator. In case of the SW implementation, the time of circuit evaluation depends on the size of the phenotype and the number of training vectors. On the other hand, in hardware, this time depends only on the number of training vectors. Hence, the accelerator becomes more useful for larger VRCs and larger sets of training data.

Table 4. The number of evaluations per second. VRC operates at 100 MHz ($dw = 4$), SW is executed on the Intel(R) Xeon(TM) CPU 3.06 GHz ($dw = 32$).

#	**VRC size (SW)**			**VRC size (HW)**			**evaluation**
inputs	**10 × 10**	**12 × 12**	**16 × 16**	**10 × 10**	**12 × 12**	**16 × 16**	**speedup**
6	400	296	222	6250	6250	6250	15–28
7	250	173	89	3125	3125	3125	12–35
8	154	95	51	1563	1563	1563	10–30
9	85	50	25	781	781	781	9–31

4.2 Symbolic Regression Problems

Similarly to the accelerator for logic circuit synthesis, the CGP accelerator for symbolic regression problems was implemented on the COMBO6X card equipped with Virtex II Pro 2VP50ff1517 FPGA. Results of synthesis are summarized in Table 5. While the PowerPC works at 300 MHz, the logic supporting the PowerPC works at 150 MHz. The remaining FPGA logic (including VRC and FU)

works at 100 MHz. Experimental results show that approximately 6,000 candidate programs can be evaluated per second when the training set consists of 15876 vectors which is 44 times faster than the same algorithm running at the Celeron 2.4 GHz [12]. This accelerator was utilized to discover novel implementations of image filters [12, 9, 13].

Table 5. Results of synthesis for the symbolic regression problems

VRC	IO blocks	BRAM	Slices	DFF
Available	852	232	23 616	49 788
4×8 CFBs	602	12	4 591	3 638
used	70%	5%	20%	7%

5 Discussion

The obtained speedup (30–40 against a common PC) is significant although only a single fitness unit was utilized. Note that the results reported in [6, 7, 4] employed multiple fitness units. In order to exploit also this level of parallelism, we can create up to 7 VRCs (depending on the number of PEs) on our FPGA. It means that the FPGA which we are currently using is able to speed up the evolution 100–200 times in comparison with a PC.

6 Conclusions

A new class of FPGA-based accelerators was presented for CGP. The accelerators contain a genetic engine which is reused in all applications. Candidate programs (circuits) are evaluated in an application-specific virtual reconfigurable circuit and fitness unit. Two types of VRCs are proposed. The first one is devoted for symbolic regression problems over the FX representation. The second one is designed for evolution of logic circuits. In both cases a significant speedup of evolution was obtained in comparison with a highly optimized software implementation of CGP. This speedup can be increased by creating multiple fitness units. Moreover, as the system is implemented on a single chip, it will be useful for online in-situ adaptive computing.

Acknowledgements

This research was partially 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 Plan No. MSM 0021630528 *Security-Oriented Research in Information Technology.*

References

[1] Koza, J.R., Keane, M.A., Streeter, M.J., Mydlowec, W., Yu, J., Lanza, G.: Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers, Dordrecht (2003)

[2] Shackleford, B.: A high-performance, pipelined, FPGA-based genetic algorithm machine. Genetic Programming and Evolvable Machines 2(1), 33–60 (2001)

[3] Tufte, G., Haddow, P.: Prototyping a GA Pipeline for Complete Hardware Evolution. In: Stoica, A., Keymeulen, D., Lohn, J. (eds.) Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware, Pasadena, CA, USA, pp. 143–150. IEEE Computer Society, Los Alamitos (1999)

[4] Martin, P.: Genetic Programming in Hardware. PhD thesis, University of Essex (2003)

[5] Fok, K.L., Wong, T.T., Wong, M.L.: Evolutionary computing on consumer graphics hardware. IEEE Intelligent Systems 22(2), 69–78 (2007)

[6] Harding, S., Banzhaf, W.: Fast genetic programming on GPUs. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 90–101. Springer, Heidelberg (2007)

[7] Chitty, D.M.: A data parallel approach to genetic programming using programmable graphics hardware. In: GECCO 2007: Proceedings of the 9th annual conference on Genetic and evolutionary computation, vol. 2, pp. 1566–1573. ACM Press, New York (2007)

[8] Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Poli, R., Banzhaf, W., Langdon, W.B., Miller, J., Nordin, P., Fogarty, T.C. (eds.) EuroGP 2000. LNCS, vol. 1802, pp. 121–132. Springer, Heidelberg (2000)

[9] Vasicek, Z., Sekanina, L.: An area-efficient alternative to adaptive median filtering in fpgas. In: Proc. of 2007 Conf. on Field Programmable Logic and Applications, pp. 216–221. IEEE Computer Society, Los Alamitos (2007)

[10] Koza, J.R., Bennett III F.H., Andre, D., Keane, M.A.: Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann, San Francisco (1999)

[11] Sekanina, L.: Evolvable components: From Theory to Hardware Implementations. In: Natural Computing, Springer, Berlin (2004)

[12] Vasicek, Z., Sekanina, L.: An evolvable hardware system in xilinx virtex ii pro fpga. International Journal of Innovative Computing and Applications 1(1), 63–73 (2007)

[13] Vasicek, Z., Sekanina, L.: Evaluation of a new platform for image filter evolution. In: Proc. of the 2007 NASA/ESA Conference on Adaptive Hardware and Systems, pp. 577–584. IEEE Computer Society, Los Alamitos (2007)

[14] Vassilev, V., Job, D., Miller, J.F.: Towards the automatic design of more efficient digital circuits. In: Proc. of the 2nd NASA/DoD Workshop of Evolvable Hardware, pp. 151–160. IEEE Computer Society, Los Alamitos, CA, US (2000)

Genetic Programming and Class-Wise Orthogonal Transformation for Dimension Reduction in Classification Problems

Kourosh Neshatian and Mengjie Zhang

School of Mathematics, Statistics and Computer Science
Victoria University of Wellington, P.O. Box 600, Wellington, New Zealand
{kourosh.neshatian,mengjie.zhang}@mcs.vuw.ac.nz

Abstract. This paper describes a new method using genetic programming (GP) in dimension reduction for classification problems. Two issues have been considered: (a) transforming the original feature space to a set of new features (components) that are more useful in classification, (b) finding a ranking measure to select more significant features. The paper presents a new class-wise orthogonal transformation function to construct a variable terminal pool for the proposed GP system. Information entropy over class intervals is used as the ranking measure for the constructed features. The performance measure is the classification accuracy on 12 benchmark problems using constructed features in a decision tree classifier. The new approach is compared with the principle component analysis (PCA) method and the results show that the new approach outperforms the PCA method on most of the problems in terms of final classification performance and dimension reduction.

1 Introduction

The quality of representation has a significant effect on the learning capability of an agent. An important issue in representation is the number of dimensions of a particular problem. In most cases, the lower the number of dimensions, the easier to learn a system. Generally with smaller number of dimensions, as long as it is enough to represent the task, the learnt models are simpler and more general. These models are also easier to interpret in most cases.

Principle component analysis (PCA) is one of the dimension reduction techniques that is widely used in different applications. The goal of PCA is to linearly transform data to a more meaningful basis. It can eliminate the redundancy between measurements (features), and reduce the noise by selecting more important components. This is done by diagonalizing the covariance matrix. However, as PCA is blind to the class labels in the training set, in many cases, it is not effective for classification problems. Another drawback of PCA is that, it makes the assumption that more diversity along an axis (feature) is a sign of a more informative and important feature and so it ranks generated components based on this factor. However, this assumption is certainly not always true.

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 242–253, 2008.
© Springer-Verlag Berlin Heidelberg 2008

The problem of dimension reduction can be seen as a feature construction problem in which the constructed features are functions of the original features and the total number of constructed features is sufficiently smaller than the number of original features in the problem. For example, the PCA method can be treated as a feature construction scenario in which all the constructed features are linear polynomials and the objective is to find the coefficients of these polynomials so that PCA goals are satisfied. A limiting issue with PCA and many other dimension reduction methods is that they have a constant general model (for example a linear polynomial) and then the dimension reduction procedure is about the estimation of the model's parameters (for example finding the coefficients of that polynomial).

Genetic programming (GP) is a flexible and expressive tool in dynamically building mathematical models based on an objective function. It means that GP expressions are not bound to any predefined template; they can be any type of expressions (linear, non-linear, trigonometric, etc) that satisfy the objective function. This feature makes GP an excellent choice for automatic feature construction. Recently there has been a new research trend in using GP for feature construction. GP has been used in different classification scenarios such as a complementary tool in the learning process of a decision tree [1], as a part of problem solving [2,3,4], or as a pre-processing phase [5].

There are two approaches to using GP for feature construction: the wrapper approach and the non-wrapper (or pre-processing) approach. In the wrapper approach [6], the final learner is used as an indicator for the appropriateness of the constructed features. In each step, the constructed features are fed into the classifier, then the classification accuracy is used as a guide to rank features [7,8]. Some approaches use a structure consisting of several program trees in each chromosome [7] while others perform a cooperative co-evolution for this purpose [8]. The number of constructed features is usually the same as the number of original features, which means that the number of dimensions cannot be reduced. However, [2] tries to compensate for this deficiency by implicitly putting pressure to select the classifiers that use a smaller number of features. Because every fitness evaluation involves a complete run of the classifier, the search process is very expensive in the wrapper approach. Moreover, as both the construction and the learning process are performed at the same time, the solutions tend to be specific to a particular type of classifier. In the non-wrapper approach, the process of feature construction is performed as a pre-processing phase. Since no particular classifier is involved in evaluating the constructed features, the process is expected to be more efficient and the results are expected to be more general. This approach can be adopted for both single feature construction [5,9] and multiple feature construction [10].

This paper aims to develop an approach to the use of GP for dimension reduction in classification problems with the goal of improving classification performance. As the vastness of the search space is a considerable issue in GP solutions, we have tried to make the search process more promising by providing a set of potential useful building blocks in a variable terminal pool. This is

achieved by proposing a new orthogonal transformation which is sensitive to the target class labels in the training data set. The original features and their transformation together comprise a variable terminal pool which is used in the GP search.

The goal of the GP search is to construct some new high level features based on the original features. Instead of wrapping a particular classifier for single feature construction as in most of the existing methods, a stand-alone fitness function based on information entropy is used for ranking a selecting constructed features. This approach is examined and compared with the standard decision tree method using the original features and using PCA transformation of the original features.

2 Using GP for Dimension Reduction

A new GP system is proposed for constructing new features and reducing the dimensionality. The overall block diagram of the system is illustrated in Figure 1. The data set with original features is divided into a training set and a test set. A proposed class-wise orthogonal linear transformation is applied to the original feature set. The details of the transformation are described in section 3. The transformed features together with the original ones are later used as the variable terminal set of the GP search. The number of dimensions is increased at this stage, however using a new ranking (fitness) function, the number of dimensions is decreased after the GP search.

The fitness function of the GP system is based on the concept of entropy over class intervals [10]. The GP search is conducted for every target class in the problem. If n is the number attributes in the original input space, and m is the number of distinct classes in the problem, then m different high level features are created as the output of the GP system. As usually m is much smaller than n, the number of dimensions of the problem is decreased at this stage. Based on the constructed features, the training and testing sets are then transformed into new training and testing sets, from which the decision tree (DT) method is used for learning. The DT classifier is then applied to the transformed test set to obtain the final classification results. The performance of the method is measured based on the classification accuracy over the test set.

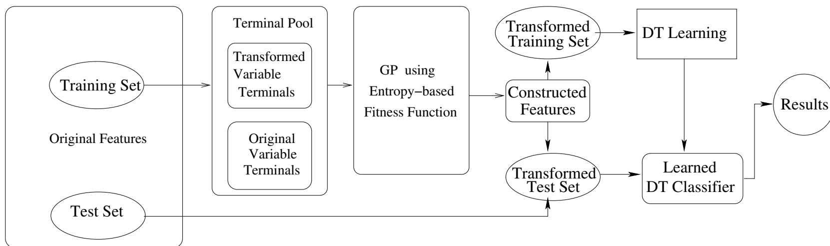


Fig. 1. Overview of the system

3 Transformation Function and Terminal Pool

3.1 DT Difficulty in Partitioning Input Space

With the assumption that instances of a particular class follow an n-dimensional normal distribution, they form a hyper-spheroid cloud around the centre of the class in a n-dimensional input space. The axes of this hyper-spheroid need not to be along the original feature space axes. This phenomenon causes many decision trees to try to cover the decision boundary with several adjacent hyper-cubes. A two dimensional example of this phenomenon is depicted in Figure 2. There are two classes with two different labels. The solid lines show the axes of distribution in each class. The dashed line shows the boundary between two classes. Because decision trees divide the input space to some rectangular sub-spaces to find the class areas, an angled class boundary causes the DT learner to make several such rectangular intervals to include the desired class instances and exclude the unwanted ones. This makes learnt trees more complicated which consequently affects their generalization capability and classification performance and also increases their execution time. This issue has been discussed in [11].

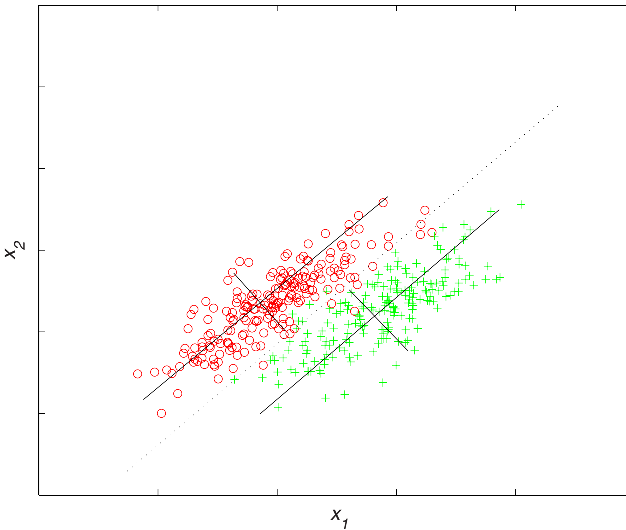


Fig. 2. An artificial data set with two attributes (x_1 and x_2 axes) and two classes (+ and $\circ$)

3.2 PCA Transformation

PCA can be applied to a data set to diagonalise the covariance matrix by linearly transforming data to a new basis that instances are distributed along new components [18]. The new location of instances are obtained by multiplying the original coordination by eigen vectors of the covariance matrix. Figure 3

shows such a transformation, in which the shape of the whole data set (including both classes) has been straightened along the new coordinates. However, because PCA cannot distinguish between different class labels, the class boundary (dashed line) is still at an angle. Training a decision tree in this new input space has similar deficiencies to the above-mentioned issue. This suggests that PCA is not suitable for dealing with the issue of angled class boundaries when using DT for classification.

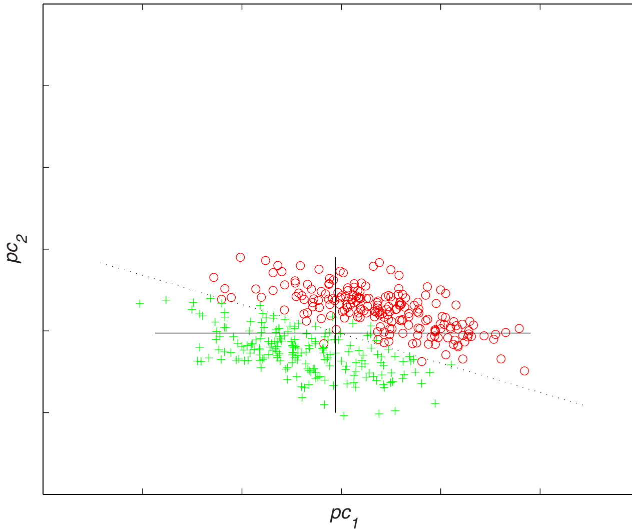


Fig. 3. Transformed input space based on the two principle components resulted by applying PCA

3.3 Class-Wise Orthogonal Transformation

To cope with this problem, we define a modified version of orthogonal transformation which is class-wise. A data set is analysed class by class, and for each class a new n-dimensional transformation is performed.

Given $\mathbf{X}$ a finite set of sequences $(x_1, x_2, \ldots, x_n)$ where $x_i \in \mathbb{R}$ and n corresponds to the input dimension of the problem and C a set of class labels $\{c_1, c_2, \ldots, c_m\}$ where m is the number of distinct classes in the problem, a training data set D can be defined as:

$$D : X \mapsto C \tag{1}$$

The training set is then divided into m partitions according to the class labels of instances. So a partition $\mathbf{P}_i$ is defined as:

$$\mathbf{P}_i = \{\mathbf{x} \mid (\mathbf{x}, c) \in D, c = c_i\} \quad where \quad \cup_{i=1}^{m} \mathbf{P}_i = \mathbf{X} \tag{2}$$

Each partition represents a hyper-spheroid cloud. We are interested in finding the axes of these hyper-spheroids (axes along which the cloud of instances are mostly scattered) which are more likely to be the boundaries of the classes. For this purpose, the covariance of data in each partition should be diagonalised. The covariance of each partition i is calculated as:

$$\mathbf{Cov}_i = E[(\mathbf{P}_i - E[\mathbf{P}_i])(\mathbf{P}_i - E[\mathbf{P}_i])^T] \tag{3}$$

where E denotes the expected value and $\mathbf{Cov}_i$ is an n by n square matrix containing the covariances (and variances along the diagonal) of features based on the data instances observed in the i-th partition. The axes of this partition (assuming data following an n-dimensional normal distribution) can be obtained by finding corresponding eigen vectors of the partition covariance matrix [18]:

$$\mathbf{A}_i = eigen(\mathbf{Cov}_i) \tag{4}$$

where each row of this matrix shows a vector in an n-dimensional space. This implies that a transformation by means of the rows of this matrix makes the axes of the resulting data set orthogonal to the new coordination system. As our goal here is to find some class boundaries that are perpendicular to the axes, we do not care about the eigen values, despite the way it is used in PCA for ranking resulting components.

While the calculation of the covariances and the eigen vectors is based on the partitioned data, the transformation is applied to the whole data set. So for each partition (distinct class in the problem), one transformation is applied. For example for the problem of Figure 2, two 2-dimensional transformations are created, one of which is shown in Figure 4. The boundary of two classes (dashed line) is now perpendicular to the new component (horizontal axis).

3.4 Terminal Pool

If n is the size of the original input space, then each transformation generates n new intermediate features. The transformation process is repeated for each class in the problem so that with m distinct class labels in a problem, $m \times n$ new intermediate features are constructed at this stage and added together with original features to the terminal set of the GP search. So the number of variables in the pool is equal to $n \times (m + 1)$. While this is an increase in the number of dimensions, it gives more chance to the GP search to find an appropriate high level feature. Later on an entropy-based fitness function is used to choose the best features among a set of constructed features and reduce the dimensionality.

3.5 Fitness Function

For ranking the constructed features, we have used a fitness function which is calculated based on the entropy of the constructed features over class interval [10]. Assuming that the class data approximately follow normal distributions, we can use equation 5 to determine the boundaries of the interval. In this equation

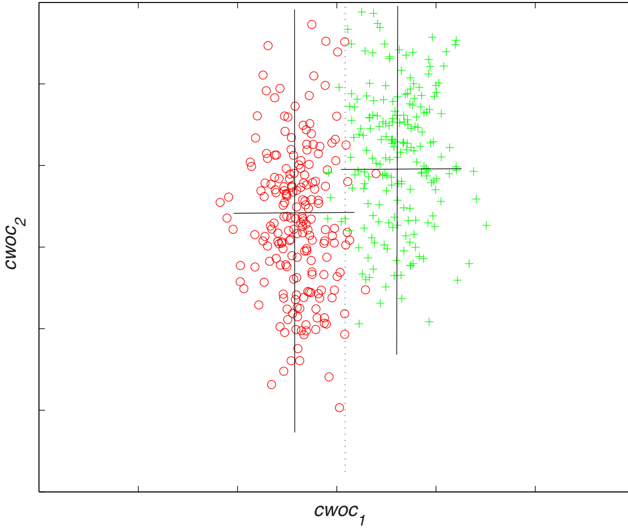


Fig. 4. Transformed input space based on the two class-wise orthogonal transformations (out of four)

x_c is the value of the feature for an instance of class c and μ and σ are mean and standard deviation of the class along the feature respectively. This interval can theoretically cover 99% of the class instances [12].

$$\mu - 3\sigma \leq x_c \leq \mu + 3\sigma \tag{5}$$

The discovered interval is regarded as an information channel and different class labels over the interval as different symbols in the channel. Accordingly, the entropy of the channel is used to measure the level of uncertainty [13] as shown in equation 6. In this equation, I is the interval, C is the set of all classes, c is the class index, and $P_I(c)$ is the probability of class c in interval I.

$$Entropy(I) = \sum_{c \in C} -P_I(c) \log_2 P_I(c) \tag{6}$$

For a training set S and a target class c, the fitness of an individual program p is calculated as follows:

1. Use program p to convert each example in the training set S to a new transformed set S_{new}. The program uses the original feature values of an instance as inputs and produces a single floating point value for each instance.
2. Make a subset Sub_c from all instances of the target class c in S_{new}.
3. Calculate the mean μ and standard deviation σ over all examples of Sub_c.
4. Determine an $Interval_c$ according to equation (5), which is $[\mu - 3\sigma, \mu + 3\sigma]$.
5. Collect all instances from S_{new} which fall in the interval and calculate the entropy over $Interval_c$ using equation 6. The entropy is used as the fitness of the program p.

If most of the instances falling into an interval belong to a single class, the entropy will be quite low. According to this design, the smaller the fitness, the better the program, and consequently the better the constructed feature. For each class in the problem, one GP run needs to be conducted. At each run the fitness function focuses on a particular class. At the end of the run, the best evolved program will be used as the constructed feature for that class. So for every problem the number of constructed features is equal to the number of classes in the problem.

4 Experiment Design and Configurations

A set of four experiments have been conducted on each data set(problem). In the first experiment the original features were fed to the DT classifier to determine the performance of classification on each data set. The results of this experiment were used as a baseline for other experiments. In the second experiment the data set was transformed using PCA and then all the generated components were fed to the DT for classification. In the third experiment, PCA was used for transforming the original data set but only high ranked components were chosen. The number of selected components was equal to the number of features generated by GP. In the fourth experiment, the proposed GP system was used for constructing features. In this experiment, a terminal pool was created based on the proposed method; then a GP search with the proposed fitness function was conducted for each target class in the problem.

A 10-fold cross-validation has been used for all the experiments. 30 different random seeds were generated for each experiment. For each generated random seed, the data set was shuffled and stratified to 10 folds. Each time one of the folds was taken as the test fold and the remaining as the training set. It means that each experiment was repeated 300 times. For the GP experiment if m is the number of classes in the problem, then $300 \times m$ runs were conducted for each problem. The performance of the DT classifier on the constructed features has been the main measure for comparing different approaches.

4.1 GP Settings

We used the tree-based structure to represent genetic programs [14], each of which produces a single floating number output. The terminal (feature) pool described in the previous section was used as the terminal set of the GP system. A number of randomly generated constants were also added to the terminal set during run time.

The four standard arithmetic operators were used to form the function set: $FuncSet = \{+, -, \times, \div\}$. The $+$, $-$, and $\times$ operators had their usual meanings — addition, subtraction and multiplication, while $\div$ represented "protected" division which is the usual division operator except that a divide by zero gives a result of zero. Each of these functions takes two arguments.

The ramped half-and-half method [14] was used for generating programs in the initial population and for the mutation operator. During the search process we used a heavy dynamic limit on tree depth [16] to control the code bloating. The probability of the crossover and mutation operators are adapted automatically at run time [17]. An elitist approach has been taken to keep the best individual of the generation. The population size was 1000. The initial maximum program tree depth was set to 3 but the trees could be increased to a depth of 6 during evolution. The evolution was terminated at a maximum number of generations of 200. In all experiments, the J48 implementation of the C4.5 decision tree inducer [11] was used to evaluate the quality of the constructed features.

4.2 Data Sets

We used 12 data sets collected from the UCI machine learning repository [15] in the experiments. Table 1 summarises the main characteristics of these data sets. These include binary and multiple class classification problems and also problems with either relatively low or high number of attributes. In some data sets, for example the original Wisconsin breast cancer data set, instances with missing values have been excluded.

Table 1. Specification of data sets used in experiments

Problem	# Features	# Instances	# Classes
Balance Scale	4	625	3
Glass Identification	9	214	6
Iris Plant	4	150	3
Johns Hopkins Ionosphere	34	351	2
Liver disorders	6	345	2
Pima Indians Diabetes	8	768	2
Sonar	60	208	2
Thyroid Disease	5	215	3
Waveform	21	5000	3
Wine Recognition	13	178	3
Wisconsin Breast Cancer (WBC)-Diagnostic	30	569	2
Wisconsin Breast Cancer (WBC)-Original	9	683	2

5 Results

The experiment results are presented in Table 2. The table has two parts for each problem. The first part shows the number of features in the different stages of the proposed GP process. The first column in this part (Org.) shows the number of original features in the problem. The second column (Pool) shows the number of generated features in the terminal pool of the GP system. It includes transformation of all the original features under the proposed transformation function for each class in the problem, plus the original feature set. For example,

Table 2. Result of standard DT, DT and PCA, and the proposed GP system

Problem	# of Features				Classification Performance (%)			
	Org.	Pool	Cnst.	Reduct. (%)	Org.	PCA	PCA-DR	GP
Balance Scale	4	16	3	25.0	78.1	90.1	85.9	**100**
Glass Identification	9	63	6	33.3	67.8	**68.8**	68.3	64.3
Iris Plant	4	16	3	25.0	95.0	94.7	94.7	**95.3**
JH Ionosphere	34	102	2	94.1	89.6	86.6	81.8	**90.1**
Liver Disorders	6	18	2	66.7	65.9	63.4	56.8	**69.5**
Pima Diabetes	8	24	2	75.0	**74.6**	74.6	72.0	74.4
Sonar	60	180	2	96.7	73.2	74.9	48.5	**78.4**
Thyroid Disease	5	20	3	40.0	93.2	87.4	86.1	**95.9**
Waveform	21	84	3	85.7	76.4	76.9	**86.2**	85.7
Wine Recognition	13	52	3	76.9	93.0	84.3	81.5	**94.8**
WBC-Diagnostic	30	90	2	93.3	93.5	92.6	93.1	**96.6**
WBC-Original	9	27	2	77.8	95.3	97.5	**97.5**	97.0
Average				65.8%	82.9%	82.7%	79.4%	**86.8%**

in Glass Identification problem, there are $(6 \times 9 + 9)$ features in the pool. The third column (Cnst.) shows the number of constructed features (output of the GP system) which is equal to the number of classes in the problem. The fourth column shows the dimension reduction ratio for each problem which is calculated by $\frac{Org.-Cnst.}{Org.}$.

The second part of Table 2 shows the classification performance of J48 decision tree on different feature sets using the test folds. The first column in this part shows the classification performance using the original feature set. The second column (PCA) shows the performance of classification when all the features are transformed by the PCA method to a new set of components. Note that this includes all the generated components which are as many as the number of original features. The third column in this part (PCA-DR) shows the classification performance when only high ranked components are selected from the PCA transformation. For the sake of comparison the number of selected components is equal to the number of generated features by the GP system. The last column shows the performance of the classifier after using GP generated features. Bold numbers in each row show the highest performance in that row.

As shown in Table 2, on all the problems, the dimensionality has been decreased by GP. However as the number of constructed features is related to the task, the reduction rate is different from one problem to another. The dimension reduction rate is over 50% on 8 out of the 12 classification problems. On average the reduction rate on all problems has been around 66%.

Comparing the classification performance achieved by constructed features (GP column of Table 2) with the original classification performance (Orig. column in that table), we found out that on 10 out of all the 12 data sets, the new system has been able to improve the classification performance while considerably reducing the number of dimensions. On 8 problems the performance of the proposed GP system outperforms all other approaches. On three tasks

(Pima, Waveform and WBC-Original), the performance of the new approach is very similar to the best performing system and only on one problem (Glass Identification) the performance has been 4.5% less than the best performing system.

When PCA is used as transformation method, only on one problem (Glass identification) it performs the best. When PCA is used as a transformation and dimension reduction tool (PCA-DR column), only on two problems (Waveform and WBC-Original) it performs the best.

The average of 86.8% for classification performance using the proposed GP system for feature construction over 12 data sets, shows a significant improvement over all other methods.

6 Conclusions

The goal of this research was to develop an approach to reducing the dimensionality of classification problems and improving the classification performance. The proposed approach was to transform the original input space to a new one via a set of GP-constructed new features. A class-wise orthogonal transformation was proposed for generating a variable terminal pool of the GP search to establish perpendicular class boundaries between classes. We used the entropy of class intervals as a fitness measure for constructed features to provide better class separation in the new transformed input space.

The approach was examined and compared with the standard decision tree approach, and the PCA combined with the decision tree approach. The results show that the proposed GP system not only achieved the initial goal of dimension reduction, but also significantly improved the classification performance in most cases. The results also show that the proposed system outperforms the standard decision tree method in terms of performance and the DT method with PCAs in terms of dimension reduction and classification performance. This suggests that GP is an effective approach to reducing the dimensionality of classification problems and improving the classification performance by constructing a set of high-level features.

For future work, we would consider providing a more flexible model to find class intervals without depending on the assuption that the class instances are normally distributed. Another development to the current approach would be providing the capability of constructing an arbitrary number of features for each problem.

Acknowledgment

This work was supported in part by the Marsden Fund (05-VUW-017) administrated by the Royal Society of New Zealand and the Research Fund (URF7/39) at Victoria University of Wellington.

References

1. Ekart, A., Markus, A.: Using genetic programming and decision trees for generating structural descriptions of four bar mechanisms. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 17(3), 205–220 (2003)
2. Muni, D.P., Pal, N.R., Das, J.: Genetic programming for simultaneous feature selection and classifier design. IEEE Transactions on Systems, Man and Cybernetics, Part B 36(1), 106–117 (2006)
3. Krawiec, K., Bhanu, B.: Visual learning by co-evolutionary feature synthesis. IEEE Transactions on System, Man, and Cybernetics – Part B 35(3), 409–425 (2005)
4. Bhanu, B., Krawiec, K.: Co-evolutionary construction of features for transformation of representation in machine learning. In: GECCO 2002. Proceedings, July 8 2002, pp. 249–254, AAAI, New York (2002)
5. Otero, F.E.B., Silva, M.M.S., Freitas, A.A., Nievola, J.C.: Genetic programming for attribute construction in data mining. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E.P.K., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 384–393. Springer, Heidelberg (2003)
6. Kohavi, R., John, G.: Wrappers for feature subset selection. In: Artificial Intelligence, pp. 273–324 (1997)
7. Smith, M.G., Bull, L.: Genetic programming with a genetic algorithm for feature construction and selection. Genetic Programming and Evolvable Machines 6(3), 265–281, Published online (August 17, 2005)
8. Krawiec, K.: Genetic programming-based construction of features for machine learning and knowledge discovery tasks. Genetic Programming and Evolvable Machines 3(4), 329–343 (2002)
9. Muharram, M.A., Smith, G.D.: Evolutionary Feature Construction Using Information Gain and Gini Index. In: Keijzer, M., O'Reilly, U.-M., Lucas, S.M., Costa, E., Soule, T. (eds.) EuroGP 2004. LNCS, vol. 3003, pp. 379–388. Springer, Heidelberg (2004)
10. Neshatian, K., Zhang, M., Johnston, M.: Feature Construction and Dimension Reduction Using Genetic Programming. In: Orgun, M.A., Thornton, J. (eds.) AI 2007. LNCS (LNAI), vol. 4830, pp. 160–179. Springer, Heidelberg (2007)
11. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann, San Francisco (1993)
12. Kreyszig, E.: Advanced Engineering Mathematics, 8th edn. John Wiley, Chichester (1999)
13. Mitchell, T.: Machine Learning. McGraw-Hill, New York (1997)
14. Koza, J.R.: Genetic Programming. MIT Press, Cambridge (1992)
15. Asuncion, A.,, D.N.: UCI machine learning repository (2007)
16. Silva, S., Costa, E.: Dynamic limits for bloat control: Variations on size and depth. In: Deb, K., al., e. (eds.) GECCO 2004. LNCS, vol. 3103, pp. 666–677. Springer, Heidelberg (2004)
17. Davis, L.: Adapting operator probabilities in genetic algorithms. In: Proceedings of the Third International Conference on Genetic Algorithms, pp. 70–79. Morgan Kaufmann, San Francisco (1989)
18. Fukunaga, K.: Introduction to statistical pattern recognition, 2nd edn. Academic Press, London (1990)

Evolving Proactive Aggregation Protocols

Thomas Weise, Michael Zapf, and Kurt Geihs

University of Kassel, Wilhelmshöher Allee 73,
D-34121 Kassel, Germany
{weise,zapf,geihs}@vs.uni-kassel.de
http://www.vs.uni-kassel.de

Abstract. We present an approach for the automated synthesis of proactive aggregation protocols using Genetic Programming and discuss major decisions in modeling and simulating distributed aggregation protocols. We develop a genotype, which is an abstract specification form for aggregation protocols. Finally we show the evolution of a distributed average protocol under various conditions to demonstrate the utility of our approach.

1 Introduction

Genetic Programming has some popular application areas like the synthesis of analog electrical circuits, cellular automata, and data mining. In larger networks, especially sensor networks or MANETs, the design of protocols for distributed processing may become a challenge. We believe that employing Genetic Programming for the automatic synthesis of protocols reveals a lot of future potential for designing distributed systems [1,2,3].

In order to unleash this potential, a clear and structured approach is needed. In this paper, we stepwise exercise such an approach on the example of automated distributed aggregation protocol synthesis.

Aggregation functions with their ability to summarize information in a certain, user-specified way are a very important building block for distributed applications [4]. We illustrate its utility in sensor networks [5] in an initial example scenario in Section 2. This example helps to clarify the problem domain and the required features of possible solutions. We then select a suitable Genetic Programming technique that can be extended to suffice these needs. The next step (taken in Section 3) is to derive suitable models for the entities in the problem domain. This means modeling the *relevant* properties of both, the nodes that will run the protocols and the network itself. Now the structure of the solution candidates and how they have to be simulated in order to determine their fitness can be defined, as done in Section 4. Section 5 shows results from experiments which demonstrate the applicability of the approach before we conclude this article in Section 6.

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 254–265, 2008.
© Springer-Verlag Berlin Heidelberg 2008

2 The Scenario

2.1 Gossip-Based Aggregation

Jelasity, Montresor and Babaoglu [6] propose a simple yet efficient type of proactive aggregation protocols [7]. Its basic assumption is that each node in a network holds one numerical value x. This value represents the information about the node or its environment that should be aggregated, for example the current work load. The task of an aggregation protocol is to provide all nodes in the network with an up-to-date estimate of the aggregate function $\alpha(\boldsymbol{x})$ of the vector of all values $\boldsymbol{x} = (x_p, x_q, \ldots)$. Of course, we cannot compute α directly since $\boldsymbol{x}$ is distributed over the network.

The nodes hold local states s (containing x) which they can exchange via messages. Therefore, each node regularly picks a communication partner with the function *getNeighbor()*. Once in each $\delta > 0$ time units, at a randomly picked time, a node p selects a neighbor q. Both partners exchange their information and update their states with the *update* method: p calls $update(s_p, s_q)$ and q invokes $update(s_q, s_p)$. *update* is defined according to the aggregate that we want to be computed.

2.2 The Distributed Average

Imagine a network of distributed temperature sensors, carrying a little display visible to the public. The temperatures measured locally will fluctuate because of wind or light changes. Thus, the displays should not only show the temperature measured by the sensor node they are directly attached to, but also the average of all temperatures measured by all nodes. The network needs to execute a distributed aggregation protocol in order to estimate that average. If we choose a gossip-based average protocol, each node will hold a state variable containing its local estimation of the mean. The *update* function, receiving the local approximation and the estimate of another node, returns the mean of its inputs.

$$update_{avg}(s_p, s_q) = \frac{s_p + s_q}{2} \tag{1}$$

If two nodes p and q communicate with each other, the new value of s_p and s_q will be $s_p(t+1) = s_q(t+1) = 0.5 * (s_p(t) + s_q(t))$. The sum – and thus also the mean – of both states remains constant. Their, variance, however becomes 0 and so the overall variance in the network gradually decreases.

2.3 Why Synthesize Aggregation Protocols?

There are three use cases for an automated aggregation protocol synthesis:

- We may already know a valid aggregation protocol but want to find an equivalent protocol which has advantages like faster convergence or robustness in terms of input volatility. This case is analogous to finding arithmetic identities in symbolic regression.

– We do not know the aggregate function α nor the protocol but have a set of sample data vectors $\boldsymbol{x}_i$ (maybe differing in dimensionality) and corresponding aggregates y_i. Using Genetic Programming we attempt to find an aggregation protocol that fits to this sample information.
– The most probable use case is that we know how to compute the aggregate locally with a given α function but want to find a distributed protocol that does the same. In order to automate this transformation, we use α to create sample data sets and then apply the approach of the second use case.

2.4 Symbolic Regression

Our goal is to provide a framework which allows such an automated translation of a (local) aggregate function α into a proactive, gossip-based aggregation protocol. For this purpose we extend Koza's technique of *symbolic regression* [8].

Regression means finding a function $f^\star : \mathbb{R}^m \mapsto \mathbb{R}$ that approximates an unknown relation φ of one dependent variable $y \in \mathbb{R}$ to m independent variables $x_1, x_2, x_3, \dots, x_m$ by analyzing a given set of sample data $S = \{(x_{1,i}, x_{2,i}, \dots, x_{m,i}, y_i)\}$. Traditional approaches like linear regression define a parametric curve $f_{\mathfrak{P}}$ and then minimize the mean square error MSE of the curve to the data samples by optimizing the parameters $\mathfrak{P}$.

$$MSE(f) = \frac{1}{|S|} \sum_{i=1}^{|S|} \left[y_i - f_{\mathfrak{P}}(x_{1,i}, \dots, x_{m,i})\right]^2 \tag{2}$$

Symbolic regression does not limit the solution to a given form, as mentioned in numerous works [8,9,10]. Here, mathematical expressions are represented as tree structures. Nodes are mathematical functions and their child nodes are their parameters. Real constants and the independent variables $x_1 \dots x_m$ act as leafs. The (functional) objective function is usually the mean square error $\gamma_1(f) \equiv MSE(f)$ which turns symbolic regression into a maximum likelihood estimation method [11].

In the following sections we will show how this approach can be extended in order to allow the evolution of proactive aggregation protocols.

3 Modeling

3.1 Network Model

An important aspect of communication is how the nodes select their partners for the data exchange. Jelasity, Montresor, and Babaoglu have shown that different *getNeighbor* methods influence the convergence speed of the protocols [6]. A suitable partner selection method leads to fast convergence, speeding up the simulations used for evaluating the protocols during the evolution significantly.

For networks $\mathcal{N}$ which have a number of nodes of $m = |\mathcal{N}| = 2^d$, we can specify an optimal selection scheme: In each protocol step t with $t = 1, 2, \dots$, we

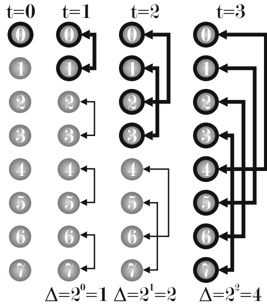


Fig. 1. pair-based dissemination

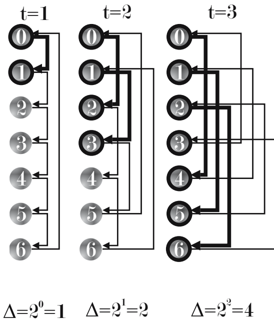


Fig. 2. generalized dissemination

Algorithm 1. $\gamma_1(u, e, r) = evalAggProtocol(u, m, T)$

Input: u, the evolved protocol *update* function
Input: m, the number of nodes in the simulation
Input: T, the maximum number of simulation steps
Output: $\gamma_1(u, e, r)$, the sum of all square errors

```
1  begin
2      d ⟵ ⌈log₂ m⌉
3      S(0) ⟵ new n × m Matrix
       // initialize with local values
4      S(0)_{I,k} ⟵ getInput(k, 0)
5      t ⟵ 1
6      while t ≤ T do
7          S(t) ⟵ copyMatrix(S(t − 1))
           // perform communication
8          k ⟵ 1
9          while k ≤ m do
10             p ⟵ (k + Δ) mod m
11             S(t)_{r_j,p} ⟵ S(t − 1)_{e_j,k} ∀ j = 1 ... |r|
12             k ⟵ k + 1
13         k ⟵ 1
14         while k ≤ m do
15             S(t)_{i,k} ⟵ getInput(k, t)
16             S(t)_{⋆,k} ⟵ u(S(t)_{⋆,k})
17             res ⟵ res + (y(t) − S(t)_{o,k})²
18             k ⟵ k + 1
19         t ⟵ t + 1
20     return res
21 end
```

compute a value $\Delta = 2^{t \bmod d}$. We then build pairs in the form $(i, i + \Delta)$, where i is the ID number of the node. This setup is optimal in terms of convergence speed, as shown in Figure 1. The data from node 0 (marked with a thick border) spreads in the first step to node 1. In the second step, it reaches node 2 directly and node 3 indirectly through node 1. In the third protocol step, the remaining four nodes receive knowledge of the information from node 0. Now the cycle would start over again.

We can generalize this approach for networks sizes that are no powers of 2. Here, we set $d = \lceil \log_2 m \rceil$ while still leaving $\Delta = 2^{t \bmod d}$ and define that a node i sends its data to the node $(i + \Delta) \bmod m$ for all i as illustrated in Figure 2.

3.2 Node Model

The model of nodes comprising a network is just as important as the network model itself. A node p executing a gossip-based aggregation protocol receives input in form of the locally known value (for example, a sensor reading) and also in form of messages containing data from other nodes in the network. The output of p is, on one hand, the local approximation of the aggregate value,

and on the other hand the information sent to its partners in the network. The computation is done by a processor which updates the local state by executing the *update* function. The local state s_p of p can most generally be represented as a vector $\boldsymbol{s}_p \in \mathbb{R}^n$ of dimension n, where n is the number of memory cells available on a node.

Until now, we have considered the states to be scalars. Generalizing them to vectors allows us to specify or evolve more complicated protocols. The state vector contains approximations of aggregate values at positions $1 \leq i \leq n$. It does not only serve as a container for the aggregate, but also as memory capable of accumulating information. It is probably unnecessary to exchange the complete state during the communication. Therefore we specify an index list e containing the indices of the elements to be sent and a list r with the indices of the elements that shall receive the values of the incoming messages. For a proper communication between the nodes, the length of e and r must be equal and each index must occur at most once in e and also at most once in r. Whenever a node p receives a message from node q, the following assignment will be done, with $\boldsymbol{s}[i]$ being the ith component of the vector:

$$\boldsymbol{s}_p[r_j] \longleftarrow \boldsymbol{s}_q[e_j] \; \forall \; j = 1 \ldots |r| \tag{3}$$

In the original form of gossip-based aggregation protocols, the state is initialized with a static input value which is stepwise refined to approximate the aggregate value [6]. In our model, this restriction is no longer required. We specify an index I pointing at the element of the state vector that will receive the input. This allows us to grow protocols for static and for volatile input data – in the latter case, the inputs are refreshed in each protocol step. A node p would then perform

$$\boldsymbol{s}_p(t)[I] \longleftarrow getInput(p, t) \tag{4}$$

The function $getInput(p, t)$ returns the input value of node p at time step t. With this definition, the state vectors $\boldsymbol{s}$ become time-dependent, written as $\boldsymbol{s}(t)$. Finally, *update* is now designed as a map $\mathbb{R}^n \mapsto \mathbb{R}^n$ to return the new state vector.

$$\boldsymbol{s}_p(t+1) = update(\boldsymbol{s}_p(t)) \tag{5}$$

In the simulation, we can put the state vectors of all nodes together to a single $n \times m$ matrix $S(t)$. The column k of this matrix contains the state vector $\boldsymbol{s}_k$ of the node k.

$$S(t) = (\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_m) \tag{6}$$

$$S_{j,k} = \boldsymbol{s}_k[j] \tag{7}$$

4 Breeding the Protocol

4.1 Evaluation and Objective Values

The models described before are the basis of the evaluation of the aggregation protocols that we breed. In general, there are two functional features that we want to develop in the artificial evolution:

1. We want to grow aggregation protocols where the deviation between the local estimates and the global aggregate is as small as possible, ideally 0.
2. This deviation can surely not be 0 after the first iteration at $t = 1$, because the nodes do not know all data at that time. However, the way how received data is incorporated into the local state of a node influences the speed of convergence to the wanted value. We want to find protocols that converge as quickly as possible.

In all use cases discussed in Section 2.3, we already know either the correct aggregation values y_i or the local aggregate function $\alpha : \mathbb{R}^m \mapsto \mathbb{R}$ that calculates them from data vectors of the length m. The objective is to find a distributed protocol that computes the same aggregates in a network where the data vector is distributed over m nodes. In our model, the estimates of the aggregate value can be found at the positions $S_{O,\star} \equiv \boldsymbol{s}_k[O] \; \forall \; k \in 1 \ldots n$ in the state matrix or the state vectors respectively.

The deviation $\varepsilon(k,t)$ of the local approximation of a node k from the correct aggregate value $y(t)$ at a point in time t denotes its estimation error.

$$y(t) = \alpha\left(\left(getInput(1,t), \ldots, getInput(m,t)\right)'\right) \tag{8}$$

$$\varepsilon(k,t) = y(t) - S_{O,k}(t) = y(t) - \boldsymbol{s}_k[O] \tag{9}$$

We have already argued that the mean square error is an appropriate quality function for symbolic regression. Analogously, the mean of the squares of the errors ε over all simulated time steps and all simulated nodes is a good criterion for the utility of an aggregation protocol. It tangents both functional aspects subject to optimization: The larger it is, the greater is the deviation of the estimates from the correct value. If the convergence speed of the protocol is low, these deviations will become smaller more slowly by time. Hence, the mean square error will also be higher. For any evolved *update* function u we define[1]:

$$\gamma_1(u,e,r) = \frac{1}{T * m} \sum_{t=1}^{T} \sum_{k=1}^{m} \varepsilon(k,t)^2 \bigg|_{u,e,r} \tag{10}$$

This rather mathematical definition is realized indirectly in Algorithm 1, which returns the value of γ_1 for an evolved *update* method u and also applies the fast, convergence-friendly communication scheme discussed in Section 3.1.

4.2 Phenotypic Representation

We have to find a proper representation for gossip-based aggregation protocols. Such a protocol consists of two parts: the evolved *update* function and a specification of the properties of the state vector – the variables I, O, r, and e.

[1] Where $|_{u,e,r}$ means "passing u, e, r as input to Algorithm 1"

Representation for the *update* Function. The function *update* as defined in the context of our basic model for aggregation protocols receives the state vectors $\boldsymbol{s}_k(t) \in \mathbb{R}^m$ of time step t as input. It returns the new state vectors $\boldsymbol{s}_k(t+1) \in \mathbb{R}^m$ of time step $t+1$. This function is indeed an algorithm by itself which can be represented as a list of tuples $l = [\dots, (u_j, v_j), \dots]$ of mathematical expressions u_j and vector element indices v_j. This list l is processed sequentially for $j = 1, 2, \dots, |l|$. In each step j, the result of the expression u_j is computed and assigned to the v_jth element of the old state vector $\boldsymbol{s}(t-1)$. In the simplest case, l will have the length $|l| = 1$. One example for this is the well-known distributed average protocol illustrated in Figure 3: In the single formula, the first element of $\boldsymbol{s}_1(t)$, [1], is assigned to $0.5 * ([1] + [2])$ which is the average of its old value and the received information. Here the value of the first element is send to the partner and the received message is stored in the second element, i.e. $r = [2], e = [1]$. The terminal set of the expressions now does not contain the simple variable x anymore but all elements of the state vectors. Finally, after all formulas in the list have been computed and their return values are assigned to the corresponding memory cells, the modified old state vector $\boldsymbol{s}_k(t)$ becomes the new one $\boldsymbol{s}_k(t+1)$.

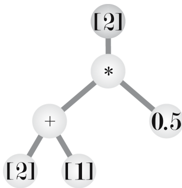


Fig. 3. A distributed average protocol

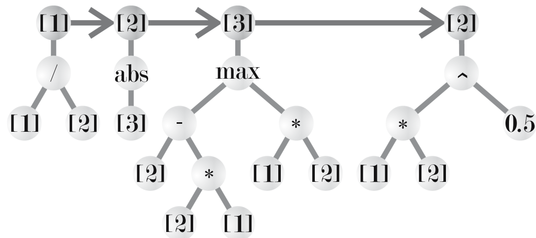


Fig. 4. The square root of the distributed average protocol

Figure 4 shows a more complicated protocol where *update* consists of $|l| = 4$ formulas $[(u_1, 1), (u_2, 2), (u_3, 3), (u_4, 2)]$. We will not elaborate deeper on these examples but just note that both are obtained with Genetic Programming. The point is that we are able to provide a form for the first part of the aggregation protocol specification that is compatible to normal symbolic regression and which hence can be evolved using standard operators.

Besides a sequence of formulas computed repetitively in a cycle, we also need an additional sequence that is executed only once, in the initialization phase. This is needed for some other protocols than the distributed minimum, maximum, and average, which cannot assume the approximation of the estimate to be the current input value. Here, another sequence of instructions is needed which transforms the input value into an estimate which then can be exchanged with other nodes and used as basis for subsequence calculations. This additional

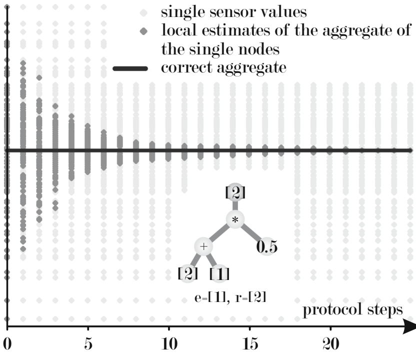


Fig. 5. The behavior of the distributed average protocol with static inputs

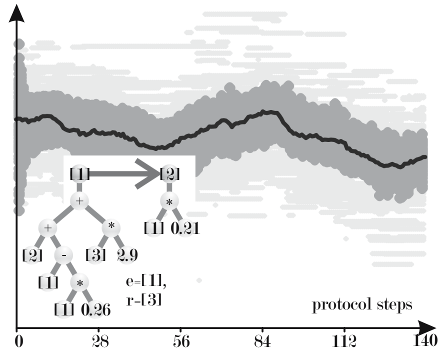


Fig. 6. A dynamic aggregation protocol for the distributed average

sequence is evolved and treated exactly in the same way as the set of formulas used inside the protocol cycle.

Straightforward, we can specify a non-functional objective function γ_2 that returns the number of expressions in both sets and hence puts pressure into the direction of small protocols with less computational costs.

Representation for I, O, e, and r. Like the *update* function, the parameters of the data exchange, r and e, become subject to evolution. I and O are only single indices; we can assume them to be fixed as $I = 1$ and $O = 2$. Although we could do the same with e and r, there is a very good reason to keep them variable: If e and r are built during the evolutionary process, different protocols with different message lengths ($|e_1| \neq |e_2|$) can emerge. Therefore, we introduce a non-functional objective function γ_3 minimizing the message lengths. The results of Genetic Programming will thus be optimal not only in accuracy of the aggregates but also in terms of communication costs. A good encoding scheme for e and r is a variable-length integer string (array) for each of the two. Such genomes are common and we can reuse standard operators of genetic algorithms.

4.3 Volatile Input Data

The specification of $getInput(k, t)$ which returns the input value of node k at time $t \in [0, T]$ allows us to evolve aggregation protocols for static as well as for volatile input. Traditional aggregation protocols are only able to deal with static inputs [6], having good convergence properties, as illustrated in Figure 5.

They would need to be restarted in a real application from time to time in order to provide up-to-date approximations of the aggregate. This approach is good if the input values in the real application change slowly. If they are volatile, the estimations of these protocols become more and more imprecise. The fact that an aggregation protocol needs a certain number of cycles to converge is an issue especially in larger or mobile sensor networks. One way to solve

this problem is to increase the data rate of the network accordingly and to restart the protocols more often. If this is not feasible, because for example energy restrictions in a low-power sensor network application prohibit increasing the network traffic, dynamic aggregation protocols may help. They represent a sliding average of the approximated parameter and are able to cope with changing input data. In each protocol step, they will incorporate their old state, the received information, and the current input data into the calculations. A dynamic distributed average protocol like the one illustrated in Figure 6 is a weighted sum of the old estimate, the received estimate, and the current value. The weights in the sum can be determined by the Genetic Programming process according to the speed with which the inputs change. In order to determine this speed for the simulations, a few real sample measurements would suffice to produce customized protocols for each application situation.

5 Results from Experiments

For our experiments, we have used a simple elitist evolutionary algorithm with a population size of 4096 and an archive size of 64. In the simulations, 16 virtual machines were running, each holding a state vector $\boldsymbol{s}$ with five elements. For evaluation, we perform 22 simulation runs per protocol where each run is granted 28 cycles in the static and 300 cycles in the dynamic case. In the evolution, we put most of the pressure on optimizing the first objective, and only take the other values into consideration to break ties. This tiered comparison structure [11] leads to optimal sets with few members that most often (but not always) have equal objective values and only differ in their phenotypes.

5.1 Average – Static

With this configuration, protocols for simple aggregates like minimum, maximum, and average can be obtained in just a few generations. We have used the distributed average protocol which computes $\alpha_{avg} = \overline{\boldsymbol{x}}$ in many of the previous examples, for instance in Section 2.2 and in Figure 4.

The evolution of a static version of such an algorithm is illustrated in Figure 7. It shows how the values of the first objective function (the mean square error sum) improve with the generations in twelve independent runs of the evolutionary algorithm. All runs actually converged to the optimal solution previously discussed, most of them very quickly in less then 50 generations.

Figure 8 reveals the inter-relation between the first and second objective function for two randomly picked runs. Most often, when the accurateness of the (best known) protocols increases, so does the number of formula expressions. These peaks in γ_2 are always followed by a recession caused by stepwise improvement of the protocol efficiency by eliminating unnecessary expressions. This phenomenon is rooted in the tiered comparison that we chose: A larger but more precise protocol will always beat a smaller, less accurate one. If two protocols have equal precision, the smaller one will prevail.

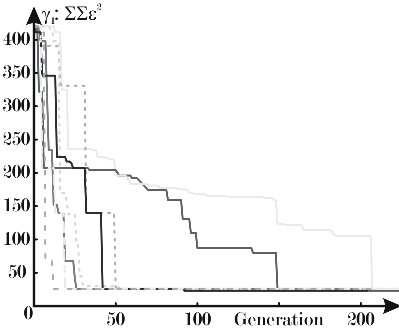


Fig. 7. The evolutionary progress of the *average* protocol

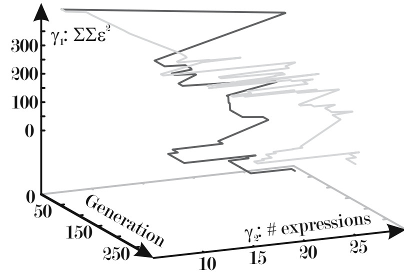


Fig. 8. The relation of γ_1 and γ_2 in the *average* protocol

5.2 Root-of-Average – Static

In [3] we used the evolution of the *root-of-average* protocol as benchmark problem. Here, a distributed protocol for the aggregate function α_{ra} shall be evolved:

$$\alpha_{ra}(\boldsymbol{x}) = \sqrt{|\overline{\boldsymbol{x}}|} \tag{11}$$

One result of these experiments has already been sketched in Figure 4. Figure 9 is a plot of eleven independent evolution runs. It also shows a solution found after only 84 generations in the quickest experiment. The values of the first objective function γ_1, denoting the mean square error, improve so quickly in all runs at the beginning that a logarithmic scale is needed to display them properly. This contrasts with the simple average protocol evolution where the measured fitness is approximately proportional to the number of generations. The reason is the underlying aggregate function which is more complicated and thus, harder to approximate. Therefore, the initial errors are much higher and even small changes in the protocols can lead to large gains in accurateness.

The example solution contains a useless initialization sequence. In the experiments, it paradoxically did not vanish during the later course of the evolution although the secondary (non-functional) objective function γ_2 puts pressure into the direction of smaller protocols. For the inter-relation between the first and second objective function, the same observations can be made as in the average protocol. Improvements in γ_1 often cause an increase in γ_2 which is followed by an almost immediate decrease.

5.3 Average – Dynamic

Dynamically changing inputs are more interesting, since creating protocols for this scenario by hand is more complicated. We first repeat the “average” experiment for two different scenarios with volatile input data. The first one is depicted with solid lines in Figure 10. Here, the *true* values of the aggregate $\alpha(\boldsymbol{x}(t))$ can

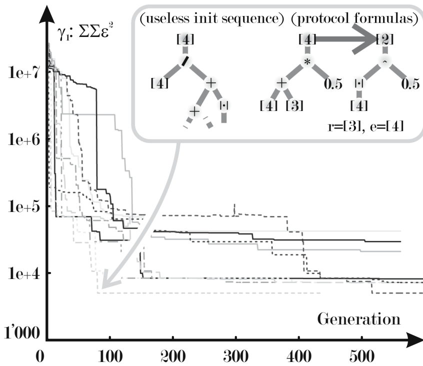


Fig. 9. The evolutionary progress and one grown solution of the *root-of-average* protocol

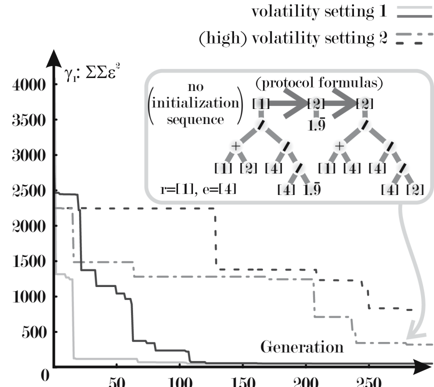


Fig. 10. The evolutionary progress of the dynamic *average* protocol

vary in each protocol step by 1% and in one simulation by 50% in total. In the second scenario, denoted by dashed lines, these volatility measures are increased to 3% and 70% respectively.

The different settings have a clear impact on the results of the error functions – the more unsteady the protocol inputs, the higher will γ_1 be, as Figure 10 clearly illustrates. The evolved solution exhibits very simple behavior: In each protocol step, a node first computes the average of its currently known value and the new sensor input. Then, it sets the new estimate to the average of this value and the value received from its partner node. Each node sends its current sensor value. This robust basic scheme seems to work fine in a volatile environment.

5.4 Root-of-Average – Dynamic

Here we follow the same approach as for the dynamic average protocol: Tests are run with the same two volatility settings as in Section 5.3. For the tests with data changing more slowly, we got a similar process of γ_1 like in Figure 9. However, we found that the evolutions with the highly dynamic input dataset did not yield functional aggregation protocols. We suppose that there is a threshold of volatility from which on Genetic Programming is no longer able to breed stable formulas.

Again, in every experiment run, increasing γ_1 is usually coupled to a deterioration of γ_2 followed by a recreation span where the formulas are reduced in size. After a phase of rest, where the new protocol supposable spreads throughout the population, this cycle starts over again until the end of the evolution.

6 Conclusions

In this article we have illustrated how Genetic Programming can be utilized for the automated synthesis of aggregation protocols. The transition to the evolution

of protocols for dynamically changing input data is a step towards a new direction. Especially in applications like large-scale sensor networks, it is very hard for a software engineer to decide which protocol configuration is best. With our evolutionary approach, different solutions could be evolved for different volatility settings which can then be selected by the network according to the current situation. The practical utilization of this new technique will be our next step in future work.

References

1. Weise, T., Geihs, K.: Genetic programming techniques for sensor networks. In: 5. GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze", Stuttgart, Germany, pp. 21–25 (2006)
2. Weise, T., Geihs, K.: Dgpf – an adaptable framework for distributed multi-objective search algorithms applied to the genetic programming of sensor networks. In: 2nd International Conference on Bioinspired Optimization Methods and their Application, BIOMA 2006, pp. 157–166. Ljubljana, Slovenia (2006)
3. Weise, T., Geihs, K., Baer, P.A.: Genetic Programming for Proactive Aggregation Protocols. In: Beliczynski, B., Dzielinski, A., Iwanowski, M., Ribeiro, B. (eds.) ICANNGA 2007. LNCS, vol. 4431, pp. 167–173. Springer, Heidelberg (2007)
4. van Renesse, R.: The Importance of Aggregation. In: Schiper, A., Shvartsman, M.M.A.A., Weatherspoon, H., Zhao, B.Y. (eds.) Future Directions in Distributed Computing. LNCS, vol. 2584, pp. 87–92. Springer, Heidelberg (2003)
5. Chong, C.-Y., Kumar, S.P.: Sensor networks: evolution, opportunities, and challenges. Proceedings of the IEEE 91(8), 1247–1256 (2003)
6. Jelasity, M., Montresor, A., Babaoglu, O.: Gossip-based aggregation in large dynamic networks. ACM Trans. Comput. Syst. 23(3), 219–252 (2005)
7. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: Proceedings of 44th Symposium on Foundations of Computer Science (FOCS 2003), Cambridge, USA, pp. 482–491. IEEE Computer Society Press, Los Alamitos (2003)
8. Koza, J.R.: Genetic Programming, On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge (1992), ISBN: 0262111705
9. Nguyen, X.H., et al.: Solving the symbolic regression problem with tree-adjunct grammar guided genetic programming: the comparative results. In: IEEE Congress on Evolutionary Computation, CEC 2002, Honolulu, USA, pp. 1326–1331 (2002)
10. Lopes, H.S., Weinert, W.R.: EGIPSYS: an enhanced gene expression programming approach for symbolic regression problems. Int. J. of Ap. Math. and Com. Sci. 14 (2004)
11. Weise, T.: Global Optimization Algorithms – Theory and Application (2007), http://www.it-weise.de/

GP Classification under Imbalanced Data sets: Active Sub-sampling and AUC Approximation

John Doucette and Malcolm I. Heywood

Faculty of Computer Science, Dalhousie University
6050 University Av.
Halifax, NS, B3H 1W5, Canada
{jdoucett,mheywood}@cs.dal.ca

Abstract. The problem of evolving binary classification models under increasingly unbalanced data sets is approached by proposing a strategy consisting of two components: Sub-sampling and 'robust' fitness function design. In particular, recent work in the wider machine learning literature has recognized that maintaining the original distribution of exemplars during training is often not appropriate for designing classifiers that are robust to degenerate classifier behavior. To this end we propose a 'Simple Active Learning Heuristic' (SALH) in which a subset of exemplars is sampled with uniform probability under a class balance enforcing rule for fitness evaluation. In addition, an efficient estimator for the Area Under the Curve (AUC) performance metric is assumed in the form of a modified Wilcoxon-Mann-Whitney (WMW) statistic. Performance is evaluated in terms of six representative UCI data sets and benchmarked against: canonical GP, SALH based GP, SALH and the modified WMW statistic, and deterministic classifiers (Naive Bayes and C4.5). The resulting SALH-WMW model is demonstrated to be both efficient and effective at providing solutions maximizing performance assessed in terms of AUC.

1 Introduction

Genetic Programming (GP) provides many unique opportunities for posing solutions to the basic Machine Learning design questions of representation, cost function, and credit assignment. In this work we are specifically interested in the topic of cost function design under the classification domain of supervised learning. Classically, an equally weighted cost function is assumed, such as 'hits' [11] or sum square error [2]. Such a design choice might be natural under balanced binary classification problems where each class carries an equal risk, but is questionable in the wider context of real world data sets that are frequently unbalanced. At the very least, as the class distribution becomes increasingly unbalanced, the likelihood of evolving degenerate classifier behavior will increase [6], [19]. Addressing the class imbalance problem has at least two related perspectives: identification of an appropriate cost (fitness) function, and sampling the original distribution of training exemplars such that the learning algorithm adapts under a different distribution than the original data set.

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 266–277, 2008.
© Springer-Verlag Berlin Heidelberg 2008

In the case of sampling algorithms, several paradigms have appeared, including: (1) boosting and bagging algorithms that tend to result in multiple individuals being built relative to static resampling of the original training data, and; (2) active learning or sub-sampling algorithms that may identify a sub-sample of exemplars from the larger training data set at each training cycle. The later case is of interest in this work. In particular we begin with the observation made from Weiss and Provost (under decision tree induction) [20]; that is, robust classifiers may be built relative to the post training performance metric of Area Under the Curve (AUC) if sub-samples are built stochastically using a uniform sampling heuristic that simultaneously enforces class balance in the sub-sample.

In this work we assume the balanced stochastic sub-sampling model as our base line model for scaling GP to larger (and therefore more interesting) data sets than would typically be the case without a hardware speedup; hereafter denoted the Simple Active Learning Heuristic (SALH). Next we investigate the utility of a fitness function capable of approximating the properties of the AUC metric. Specifically, AUC represents a rank based performance metric that explicitly measures performance in terms of two typically 'conflicting' performance goals at multiple performance points. As such, the model is encouraged to, for example, maximize recall while simultaneously minimizing false positive rate, thus explicitly penalizing degenerate behaviors that might dominate models trained from unbalanced distributions of exemplars. One drawback associated with the wider utility of AUC as a cost function in Machine Learning has been the computational cost of first estimating the Receiver Operating Characteristic (ROC) curve and then deriving the associated AUC. Naturally, by assuming a sub-sampling model we decouple the evolutionary cycle from the original dimension of the data set. However, even under such conditions a significant overhead still exists in the inner loop if we attempt to estimate the AUC directly. The final component of the model investigated in this work is therefore to make use of the Wilcoxon-Mann-Whitney (WMW) statistic where this provides a direct estimator for the AUC metric [9], [21]. To this end, we detail modifications necessary to focus the ensuing GP classifiers, such that 'robust' performance under the WMW metric was generalized to corresponding behavior under test conditions.

The proposed model of WMW fitness function estimated over exemplar subsets identified under SALH, is benchmarked over six unbalanced data sets from the UCI repository [16]. Comparisons are made against both deterministic classifiers (C4.5 and Naive Bayes), canonical GP, and GP under SALH (both of the latter assume 'hits' based fitness). The WMW model is the most successful in maximizing the area under the curve performance statistic on test data, bettering C4.5 on five of the six data sets, and significantly better than either alternative GP paradigm.

2 Related Work

As indicated in the introduction we approach the problem of designing a 'robust' classifier using two inter-related concepts: establishing a suitable training

exemplar sampling algorithm, and establishing an appropriate cost (fitness) function. Within the context of Genetic Programming in particular, several works have proposed approaches to the training exemplar sub-sampling problem. The work of Gathercole and Ross in particular demonstrated that not all exemplars are equally relevant to the training task at any point in time [8]. Two heuristics, denoted exemplar 'age' and 'difficulty' were used to bias the selection of exemplars to appear in the current fitness evaluation (training epoch). Such a model provides a considerable speedup relative to fitness evaluation over all training exemplars and was demonstrated to result in individuals performing no worse. Recent research has considered the utility of competitive coevolutionary models as the basis for an alternative model of active learning. In particular the host-parasite model of Hillis demonstrated that such a model could provide the basis for biasing the selection of a subset of training exemplars at fitness evaluation [10]. The host-parasite model does however suffer from the problem of establishing the relevant problem dependent 'virulence factor' to ensure that the exemplars selected (parasite) do not dominate the ability of the learners (host) [3], [15]. More recently, the competitive coevolutionary model for rewarding the ability of exemplars to 'distinguish' between learners under a Pareto model of coevolution has received a lot of interest [7], [17], [5]. Attempts to make use of the Pareto competitive coevolutionary paradigm under the GP classification domain have utilized a two population model, with one population representing the subset of training exemplars on which fitness evaluation is conducted, and a second population in which classifiers are evolved. Under such an architecture, Pareto competitive frameworks to date concentrate on establishing archiving strategies that posses desirable properties (such as monotonic progress) [4]. However, the indexing of exemplars by the 'point' population does not hold any implicit structure to guide the definition of appropriate variation operators. As such the most successful Pareto Competitive models, under the GP classification domain, have relied on the uniform selection of exemplar indexes and a class balancing heuristic to create the point population [13].

With respect to the utility of performance metrics that explicitly reward the evolution of 'robust' as opposed to naive classifier behaviors, many authors have considered cost functions which make use of fixed penalty functions [19], [18]. Adaptive cost functions have also been proposed, for example Eggermont *et al.* developed a scheme for periodically re-weighting the error associated with training exemplars during training [6]. This is naturally related to the 'difficulty' heuristic devised by Gathercole, but without attempting to use this as an exemplar selection bias under an active learning paradigm. Langdon and Buxton considers the problem of AUC optimization given two previous classifiers with different ROC profiles [12]. However, the problem addressed is naturally distinct from designing the initial classifiers such that ROC profiles are suitably distinct.

The two themes central to the method adopted in this work result from the findings of Weiss and Provost on training decision tree induction classifiers under unbalanced data sets [20], and a successful attempt at constructing an AUC type cost function for training a neural network classifier on a very unbalanced data

set [21]. In the case of Weiss and Provost, a systematic study is performed on the impact of training subset class balance under the C4.5 algorithm. A clear statistically significant preference was demonstrated for uniform exemplar selection while enforcing equal representation of major and minor classes. Such a heuristic was central to establishing effective operation of point population sampling algorithms for Pareto competitive coevolution of classifiers, and in some cases may out perform this model [14]. The work of Yan *et al.*, began by formally demonstrating that minimizing metrics such as mean square error or cross-entropy is not sufficient for maximizing AUC [21]. They then make use of the WMW estimator for AUC and derive an alternative, back-propagation compatible version of the metric, thus enabling them to train a multi-layer perceptron to maximize the AUC performance metric directly, and demonstrate the utility of such an algorithm under a very unbalanced 'churn' prediction problem.

3 Methodology

The basic goal of this work is to provide a generic model for the evolution of GP classifiers under unbalanced data sets through a combined approach of class balanced stochastic sub-sampling and a modified WMW cost function. The combined approach is necessary to: (1) actively bias the distribution of exemplars over which learning is conducted; (2) establish a 'robust' cost function, and; (3) address the computational cost of fitness evaluation. In the following we will define the sub-sampling based active learning model of GP classification, and WMW cost function and associated modification for the case of GP.

3.1 Simple Active Learning Model

In order to decouple the cost of fitness evaluation from the size of the training data set, an active learning model is assumed. The Dynamic Subset Selection (DSS) model of Gathercole and Ross has been widely used in the GP domain. However, in this work we assume a simpler model. Specifically, exemplars are selected with uniform probability from the original training partition, such that major and minor class provide an exemplar subset of fixed size with equal representation of both classes. The DSS algorithm was originally compared against subsets formed from exemplars sampled with purely uniform probability, but without the requirement for equal class balance [8]. This may naturally result in subsets being formed that represent major and minor classes with the same distribution as in the entire training partition. However, as the distribution of major to minor class increases, the likelihood of building 'degenerate' subsets increases, see for example the comparison of DSS and canonical GP in [13]. The study of Weiss and Provost establishes that such a scheme for building subsamples will result in optimizing for an accuracy based performance metric, but relative to a more informative performance metric such as AUC, will result in very low scores. Thus, this study adopts a class balance enforcing subsampling model that selects exemplars with uniform probability from major and minor class, until the equal class constraint is satisfied.

Aside from the active learning model for defining a new subset of exemplars at each generation, the GP classifier takes the form of canonical tree structured GP. Specifically, in the case of the wrapper operator, a sigmoid is employed where this has the desirable property of encouraging exemplars to move away from the switching point of the class boundary.

$$y(x) = \frac{2}{(1 + \exp(-GPout(x)))} - 1 \tag{1}$$

where *GPout(x)* denotes the 'raw' scalar value returned by the root node of the phenotype following evaluation of the program on input vector 'x', and $y(x)$ denotes class membership over the interval [-1, 1] with respect to exemplar 'x'. Naturally, values tending towards '-1' indicate out of class and values tending towards '1' are indicative of in class membership.

3.2 Wilcoxon-Mann-Whitney Fitness Function

The area under the curve (AUC) metric expresses classifier performance in terms of the area under the 'receiver operating characteristic' or ROC. Such curves typically characterize classifier performance in terms of true positive rate versus false positive rate [1], [9]. Unlike most widely utilized performance metrics, such as accuracy or precision and recall, the ROC curve does not rely on a single performance point to characterize classifier behavior. That is to say, both true positive rate and false positive rate are estimated at multiple performance points for each exemplar; where the performance points are derived, for example, from cuts taken across the class membership function of (1). Needless to say, the more thresholds utilized, the more accurate the characterization, but the more expensive the evaluation. Specifically, estimating the ROC curve requires the re-evaluation of true and false positive rates for a sufficient number of performance points to provide an accurate rendition of the curve. Only with this complete can estimate of the AUC. All of this takes place within the inner loop of GP. Thus in this work, we do not estimate the AUC or ROC, but make use of the Wilcoxon-Mann-Whitney (WMW) statistic, where this is already known to be an equivalent estimator for AUC without building the ROC [9]. The WMW statistic has the form,

$$WMW(I, P, N) = \sum_{i=0}^{|P|} \sum_{j=0}^{|N|} C(y, P_i, N_j) \tag{2}$$

where

$$C(y, a, b) = \begin{cases} 1 & if\, y(a) > y(b) \\ 0 & otherwise \end{cases} \tag{3}$$

and $y(a)$ is the class membership returned by the wrapper operator of equation (1) under the exemplar represented by input vector a, P is the set of all majority class exemplars, and N is the set of all minority class exemplars. Thus, $P_i(N_j)$ is the ith (jth) element of $P(N)$.

Naturally, equation (2) conducts a series of pairwise comparisons between in and out of class exemplars, only rewarding cases in which the class membership function for in class exemplars exceeds that for the out of class cases as per (3). However, when used in combination with a continuous valued (as opposed to binary) membership operator it is also necessary to explicitly reward class membership values that fall on the relevant side of the origin. We denote the resulting WMW based fitness function $WMW_{fitness}$, expressed as follows,

$$WMW_{fitness}(y, P, N) = f(y, P) \cdot f(y, N) + WMW(y, P, N) \quad (4)$$

where

$$f(y, S) = \sum_{i=0}^{|S|} \begin{cases} 1 & if\, d(S_i) = y(S_i) \\ 0 & otherwise \end{cases} \quad (5)$$

and $d(S_i)$ is the desired class label of exemplar i, and $I(S_i)$ is the corresponding binary class label suggested by the GP classifier (i.e. thresholding the wrapper about the origin).

The first component of the right hand side of (4) contributes a 'point' for every pair of correctly labeled exemplars; whereas the second component takes the form of the original WMW metric. On an exemplar by exemplar basis, the WMW contribution is satisfied first; thus, evolution will first find individuals with good AUC properties and then normalize the pairwise dominance property relative to the origin of the activation function, equation (1).

4 Results

4.1 Canonical GP

The empirical evaluation is conducted utilizing a common canonical tree structured model of GP [11] using the 1.1 distribution of lilgp [22], although the GP representation itself has no impact on the algorithm proposed. The selection operator takes the form of Koza's 'overselection', thus the top thirty two (bottom sixty eight) percent of the population account for eighty (twenty) percent of the parents. Such a model naturally has a higher take-over rate than would be the case for fitness-proportionate selection alone. The terminal set was limited to indexing the features of the problem domain, whereas the function set took the form of the four arithmetic operators, four higher order operators with a single argument ($sine(a)$, $cos(a)$, e^a, $\sqrt{a}$) and the standard conditional statement with four arguments (c if $a < b$, d otherwise). The remaining GP parameters take the form: Population size (800), Max tree nodes (256), Half-half initialization (2-6 node depth), Crossover (0.7), Mutation (0.3), Internal versus leaf node likelihood (0.9/0.1). In no cases was any attempt made to optimize these values.

4.2 Data Sets

A total of six data sets were employed in the evaluation, five corresponding to a subset of those used in the study of Weiss and Provost [20], and the sixth

corresponding to the widely utilized BUPA liver diagnosis data set. All data sets are available through the UCI repository [16] and have been selected on account of the resulting varied ratio of major to minor class distributions. As per the the Weiss and Provost study, we make the multiclass data sets binary by defining the minor class as in class and the remaining classes as the out of class exemplars. In each case the data set is stratified, with twenty five percent of the data set being withheld for the purpose of establishing a test set and the remainder of the data representing the training set. Table 1 establishes the basic properties each data set as a whole.

Table 1. Data Set Characterization

Name	Size	% Minority Class	# Attributes
Abalone	4,177	8.7	8
Sick Thyroid	3,163	9.3	25
Opt. Digits	5,620	9.9	64
Solar Flare	1,389	15.7	10
Adult	48,842	23.9	14
Liver	345	42	6

4.3 Evaluation

Evaluation is conducted in the form of three separate comparisons. In the first case we compare canonical GP with the Simple Active Learning Heuristic (SALH) of Section 3.1 over the four smaller data sets (too computationally expensive to apply canonical GP to the Adult data set). The only difference between the two models is the set of exemplars utilized for fitness evaluation. Experiment two compares GP classifiers evolved using the SALH and a count based fitness function, versus the same active learning heuristic, but with fitness evaluated over the modified WMW metric of (4). Our last comparison compares GP models evolved under the WMW metric with those trained under deterministic machine learning algorithms.

All post training evaluation will be performed in terms of test set performance as measured by the AUC metric derived from the trapezoidal integration algorithm [1]. That is to say, the AUC metric expresses the area under the curve as estimated from the receiver operating characteristic (ROC). The ROC is constructed from the performance of each classifier under true positive and false positive rates taken from twenty two points representing thresholds taken uniformly across the interval of the wrapper operator, equation (1). This results in a scalar characterization of performance, with values in the range of zero (no better than guessing) to a half (perfect classification of both minority and major classes).

Canonical GP Versus SALH. In order to establish whether the baseline canonical model of classification, that is fitness evaluation over the entire set of

training exemplars, produces classifiers that are any more robust than fitness evaluation over the balanced uniform model of subset selection, we compare post training AUC performance over fifty runs of each model on all but the 'Adult' data set from Table 1. In no cases was a statistically significant difference recorded at a Confidence interval of ninety five percent (Student T-test). Thus no negative impact is attributed to fitness evaluation conducted over exemplar subsets identified under SALH versus all training exemplars.

Combining SALH with WMW Fitness. The next test establishes the significance of introducing the WMW fitness function in combination with SALH. In effect we how have an efficient mechanism for evolving individuals under a 'robust' estimator of fitness, albeit only over subsets selected stochastically under the balance enforcing heuristic. Figures 1 and 2 summarize AUC returned on each test data set as first quartile, median, and third quartile (statistic collected over fifty runs). The WMW fitness function yields solutions with statistically significantly better AUC values under five of the six data sets at a confidence of ninety nine percent, and at ninety five percent in the case of the liver data set. The amount of variation in results returned in models trained using the WMW based fitness function are also much lower than that under the hits based fitness function. In short, even when the natural distribution of the original data set tends to equal representation of both classes, the WMW based fitness function is much more effective at directing the credit assignment process.

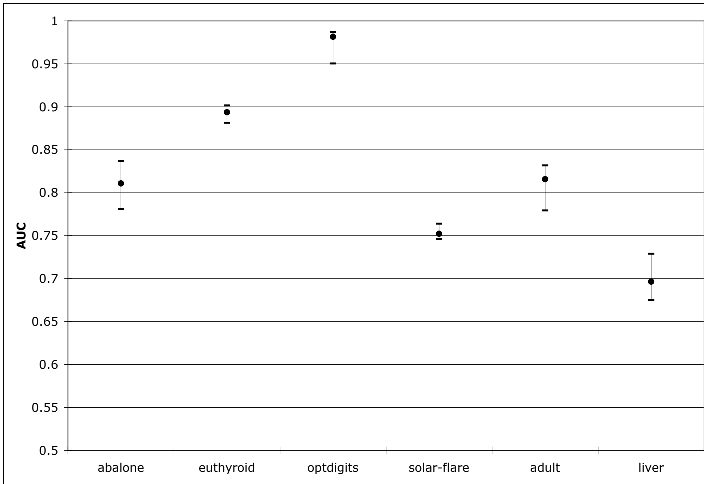


Fig. 1. SALH with WMW fitness function: Post training AUC performance under test partition for each data set

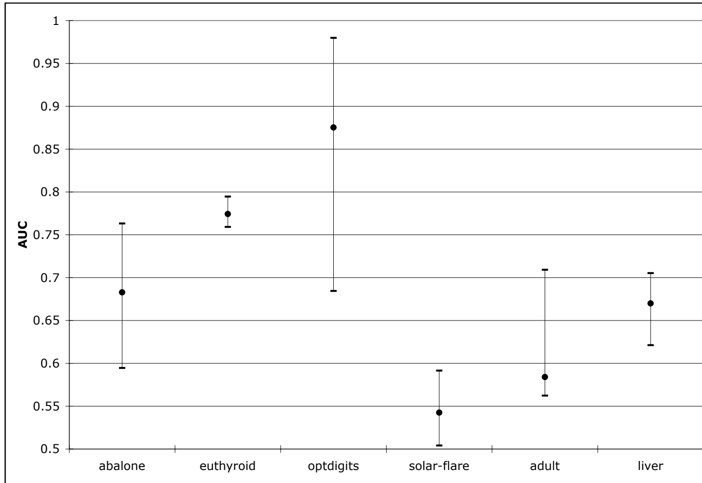


Fig. 2. SALH with 'hits' fitness function: Post training AUC performance under test partition for each data set

Comparison with Deterministic Models. Our final test compares the test set performance of models trained using Naive Bayes and C4.5 to those evolved under the WMW fitness function. This immediately presents the problem of establishing a framework for making the comparison. In particular, the deterministic models are trained following a single pass through the entire training data set, whereas evolutionary models are evolved over multiple runs. Making multiple folds of the original partition does not alter this relationship. Each fold would still require multiple runs of the evolutionary method. In effect GP requires the initialization of "free parameters" that are distinct from the learning parameters, whereas deterministic models such as C4.5 and Naive Bayes only have learning parameters. Thus, we adopt the following policy in which the deterministic model is used to establish a performance threshold against which we then ask what is the likelihood of the evolutionary model matching or bettering this performance.

Figure 3 reports the likelihood of the GP classifier initializations matching or bettering the performance of the Naive Bayes and C4.5 deterministic classifiers. Larger bars imply more of the GP solutions matched or bettered the base line established by the deterministic model. Conversely, no bar implies that all GP solutions were worse than the deterministic base line. The data set that returned no benefit from the GP model was the largest data set, Adult, where this might be an indicator for evolving over more generations (the common training limit of fifty generations implies that only seven percent of the Adult training data is sampled). Both Abalone and Euthyroid, the two data sets with the largest degree of class imbalance were most likely to result in the GP model improving on the deterministic classifier base line. Interestingly, C4.5 found the Solar flare data set particularly difficult, whereas Naive Bayes did not perform as well on

the Liver data set (the most balanced data set considered). However, the Naive Bayes classifier bettered both C4.5 and GP on the Optical Character recognition problem. In short the GP solutions were better than the Naive Bayes classifier in a minimum of sixty percent of the cases in four of the six data sets, and unable to better Naive Bayes on the other two data sets. Under the performance target set by C4.5, GP was better at least fifty percent of the time under four of the six data sets, and returned results that were better in at least twenty five percent of the initializations on the fifth data set.

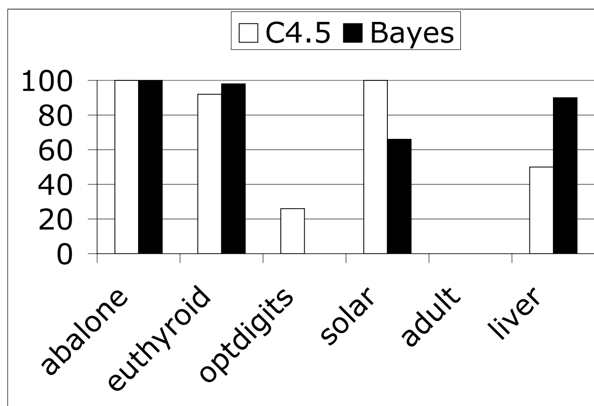


Fig. 3. Percent of SALH-WMW solutions matching or bettering the deterministic classifier baseline under post training AUC performance statistic evaluation of test partition

5 Conclusion

The problem of training GP classifiers under large unbalanced binary data sets is addressed through the dual approach of training exemplar selection and appropriate fitness function design. We begin by utilizing a class balance heuristic under an active learning paradigm, for the evolution of GP classifiers. The ensuing Simple Active Learning Heuristic is shown to perform at least as well as canonical GP evolved over all training exemplars. The second part of our approach begins with the WMW estimator for the AUC metric. As is, this metric rewards pairwise dominance behaviour as measured between minor and major class exemplars. However, we are also interested in maximizing the separation between the two sets of behaviors as mapped to 'GPout' and resolved in terms of a smooth wrapper operator, a sigmoid. To this end, we introduce a second factor into the fitness function, such that individuals that both establish the pairwise dominance property and enforce class membership relative to the wrapper operator origin receive more reward than those establishing the dominance property alone. Benchmarking on six data sets from the UCI repository with minor class

representations in the range of five to forty percent of the data set demonstrates that the proposed approach is significantly better than classifiers evolved under the same active learning heuristic and typically better than C4.5 or Naive Bayes under five and four of the six data sets respectively.

Future work will continue to investigate the significance of fitness functions in GP classifier design. In particular, recent work in machine learning has demonstrated a bias between classes of cost function and classifier operation. We anticipate there being equivalent relationships between function set design and classes of fitness function.

Acknowledgments

The authors gratefully acknowledge the support of NSERC USRA and Discovery Grant programs, and CFI New Opportunities infrastructure program.

References

1. Bradley, A.P.: The use of the area under the ROC curve in the evaluation of machine learning algorithms. Pattern Recognition 30(7), 1145–1159 (1997)
2. Brameier, M., Banzhaf, W.: A comparison of linear Genetic Programming and Neural Networks in medical data mining. IEEE Transactions on Evolutionary Computation 5(1), 17–26 (2001)
3. Cartlidge, J., Bullock, S.: Learning lessons from the common cold: How reducing parasite virulence improves coevolutionary optimization. In: IEEE Congress on Evolutionary Computation, pp. 1420–1425 (2002)
4. de Jong, E.D.: A monotonic archive for pareto-coevolution. Evolutionary Computation 15(1), 61–94 (2007)
5. de Jong, E.D., Pollack, J.B.: Ideal evaluation from coevolution. Evolutionary Computation 12(2), 159–192 (2004)
6. Eggermont, J., Eiben, A.E., van Hermert, J.I.: Adapting the fitness function in GP for data mining. In: Langdon, W.B., Fogarty, T.C., Nordin, P., Poli, R. (eds.) EuroGP 1999. LNCS, vol. 1598, pp. 193–202. Springer, Heidelberg (1999)
7. Ficici, S.G., Pollock, J.B.: Pareto optimality in coevolutionary learning. In: European Conference on Artificial Life, pp. 316–325 (2001)
8. Gathercole, C., Ross, P.: Dynamic training subset selection for supervised learning in genetic programming. In: Davidor, Y., Männer, R., Schwefel, H.-P. (eds.) PPSN 1994. LNCS, vol. 866, pp. 312–321. Springer, Heidelberg (1994)
9. Hand, D.J.: Construction and Assessment of Classification Rules. John Wiley, Chichester (1997)
10. Hillis, W.D.: Co-evolving parasites improve simulated evolution as an optimization procedure. In: Artificial Life II. Santa Fe Institute Studies in the Sciences of Complexity, vol. X, pp. 313–324 (1990)
11. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
12. Langdon, W.B., Buxton, B.F.: Evolving Receiver Operating Characteristics for Data Fusion. In: Miller, J., Tomassini, M., Lanzi, P.L., Ryan, C., Tetamanzi, A.G.B., Langdon, W.B. (eds.) EuroGP 2001. LNCS, vol. 2038, pp. 87–96. Springer, Heidelberg (2001)

13. Lemczyk, M., Heywood, M.I.: Training binary GP classifiers efficiently: a Pareto-coevolutionary approach. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 229–240. Springer, Heidelberg (2007)
14. Lichodzijewski, P., Heywood, M.I.: Pareto-coevolutionary Genetic Programming for problem decomposition in multi-class classification. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), vol. 1, pp. 464–471 (2007)
15. McIntyre, A.R., Heywood, M.I.: Toward co-evolutionary training of a multi-class classifier. In: Proceedings of the Congress on Evolutionary Computation (CEC), vol. 3, pp. 2130–2137 (2005)
16. Newman, D.J., Hettich, S., Blake, C.L., Merz, C.J.: UCI repository of machine learning databases (1998), http://www.ics.uci.edu/~mlearn/mlrepository.html
17. Noble, J., Watson, R.A.: Pareto coevolution: Using performance against coevolved opponents in a game as dimensions for Pareto selection. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), pp. 493–500 (2001)
18. Patterson, G., Zhang, M.: Fitness functions in Genetic Programming for classification with unbalanced data. In: Orgun, M.A., Thornton, J. (eds.) AI 2007. LNCS (LNAI), vol. 4830, pp. 464–471. Springer, Heidelberg (2007)
19. Song, D., Heywood, M.I., Zincir-Heywood, A.N.: Training Genetic Programming on half a million patterns: An example from anomaly detection. IEEE Transactions on Evolutionary Computation 9(3), 225–239 (2005)
20. Weiss, G.M., Provost, R.: Learning when training data are costly: The effect of class distribution on tree induction. Journal of Artificial Intelligence Research 19, 315–354 (2003)
21. Yan, L., Dodier, R., Mozer, M.C., Wolniewicz, R.: Optimizing classifier performance via the Wilcoxon-Mann-Whitney statistic. In: Proceedings of the International Conference on Machine Learning, pp. 848–855 (2003)
22. Zoungker, D., Punch, B.: lilgp genetic programming system. version 1.1 (1998), http://garage.cse.msu.edu/

Exposing a Bias Toward Short-Length Numbers in Grammatical Evolution

Marco A. Montes de Oca

IRIDIA, CoDE, Université Libre de Bruxelles, Brussels, Belgium
mmontes@ulb.ac.be

Abstract. Many automatically-synthesized programs have, like their hand-made counterparts, numerical parameters that need to be set properly before they can show an acceptable performance. Hence, any approach to the automatic synthesis of programs needs the ability to tune numerical parameters efficiently.

Grammatical Evolution (GE) is a promising grammar-based genetic programming technique that synthesizes numbers by concatenating digits. In this paper, we show that a naive application of this approach can lead to a serious number length bias that in turn affects efficiency. The root of the problem is the way the context-free grammar used by GE is defined. A simple, yet effective, solution to this problem is proposed.

1 Introduction

Genetic Programming (GP) [1] has been used for the automatic synthesis of computer programs and other kinds of systems. In many cases, a GP system is required to find two equally important components: a system's structure and optimal (or near-optimal) values for numerical parameters. Although both components determine the overall system's performance, it is its structure that determines the number and importance of numerical parameters [2]. Obviously, tuning numerical variables effectively and efficiently is crucial in any GP approach and the topic has been the subject of active research [3], [4], [5].

Previous work on the ability of Grammatical Evolution (GE) [6], a grammar-based GP technique, to synthesize numerical values has shown that a simple digit concatenation approach is superior to the traditional expression-based one [7]. In this paper, a study on the efficiency with which this approach is able to generate numerical parameters is presented. The study relies on the assumption that a good structure (i.e., the number of numerical variables) has already been found during evolution, so that the efficiency with which GE can tune numerical variables can be studied in detail.

The main finding reported here is that the classical digit concatenation grammar used by GE to generate numerical parameters induces a bias toward short-length numbers. This bias can affect substantially the efficiency of the search process which can hinder the applicability of GE as a whole. A simple, yet effective, solution to this problem is proposed. It consists of a grammar modification

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 278–288, 2008.
© Springer-Verlag Berlin Heidelberg 2008

that makes the distribution of lengths in the initial population more uniform, effectively making the search for numbers of different lengths more efficient.

The paper is organized as follows. Section 2 describes the GE approach. A brief summary of related work is presented in Section 3. Section 4 presents the number length bias problem. Section 5 describes the experimental setup used to evaluate both the magnitude of the problem and the benefits obtained with the proposed solution. The paper is concluded in Section 6.

2 Grammatical Evolution

Grammatical Evolution (GE) [6] is a recent evolutionary computation technique for the automatic synthesis of programs in an arbitrary language. At the core of the approach is a grammar-based mapping process that transforms a number of variable-length integer vectors into syntactically correct programs. The elements of an integer vector are used to select a production rule from a grammar defined in a Backus-Naur form. By expanding production rules in this way, a complete program can be generated.

The components of a solution vector are normally integers in the range $[0, 255]$. Their values are used to select a production rule[1] from the nonterminal symbol that is being expanded. The selected production rule is determined by

$$\text{selected rule} = (\text{integer value})\, \mathit{mod}\, (\text{No. of rules for current nonterminal})\,, \quad (1)$$

where *mod* denotes the modulus operator.

As an example of the operation of GE, consider the problem of performing symbolic regression. A grammar $G = \{N, T, S, P\}$ for this problem is shown below (taken from [8]). N is the set of nonterminal symbols, T is the set of terminal symbols, S is the start symbol, and P is the set of production rules.

$$\begin{aligned}
N &= \{< expr >, < op >, < func >, < var >\} \\
T &= \{\sin, \cos, \exp, \log, +, -, /, *, x, 1.0, (,)\} \\
S &= < expr > \\
P &= \{ \\
< expr > &\rightarrow < expr > < op > < expr > \\
&\quad | \, (< expr >) \\
&\quad | \, < func > (< expr >) \\
&\quad | \, < var > \\
< op > &\rightarrow + \mid - \mid / \mid * \\
< func > &\rightarrow \sin \mid \cos \mid \exp \mid \log \\
< var > &\rightarrow 1.0 \mid x \\
&\quad \}
\end{aligned}$$

[1] Rules are numbered starting from 0.

Suppose that the solution vector we want to map is

$$[6, 25, 120, 58, 43, 62, 126, 87, 67, 23, 11, 2] .$$

From the start symbol $< expr >$, there are 4 rules to choose from. Since the first element of the solution vector is 6, the selected rule is rule number $6 \, mod \, 4 = 2$. After this first expansion, the solution takes the form $< func > (< expr >)$. The mapping process continues by selecting a production rule from the leftmost nonterminal symbol, which in our example is $< func >$. In the next expansion step, the selected rule is rule number $25 \, mod \, 4 = 1$, so the solution takes the form $\cos(< expr >)$. If we continue with this process the final solution would be $\cos(\log(\exp(x))/1.0)$.

The mapping process is repeated until a string with no nonterminal symbols is generated or until no more elements in the vector remain to be mapped. If after processing all the elements of the solution vector a valid solution is still incomplete, there are two possible actions to take. The first one is called *wrapping* and consists in reinterpreting the solution vector again starting from the first element until a valid solution is generated or a maximum number of wrappings occur. Although the elements of the solution vector are reused, their effect on the generated string depends on the nonterminal symbol that is being rewritten. The second option is to discard the solution and assigning it the lowest fitness value.

By the way GE is designed, it is possible to separate the search and solution spaces. This has the advantage of decoupling the way search is done from the way solutions are constructed. Consequently, GE does not necessarily rely on genetic algorithms to work.

GE has been used in fields such as financing [9], combinatorial optimization [10] and machine learning [11]. In these and other cases a common problem stands out: synthesizing numerical values effectively and efficiently. Previous work on this direction is presented below.

3 Constant Creation by Grammatical Evolution

There have been some previous studies on the ability of GE to synthesize numbers. O'Neill et al. [7] presented a comparison between the traditional expression-based approach used in tree-based Genetic Programming with a digit concatenation one. Based on experimental evidence, they conclude that the digit concatenation approach is superior to the expression-based one on the problem of synthesizing numbers. In Dempsey et al. [12], a comparison between the digit concatenation approach and another one using random constants was performed on problems similar to those used by O'Neill et al. [7]. No conclusive evidence was found on the superiority of any of these approaches. Recently, a more detailed study was undertaken by Dempsey et al. [13] in which they finally conclude that the digit concatenation approach is superior to the random constants approach on problems requiring the synthesis of static constants. The random constants

approach proved to be better suited for dynamic problems (in which the target number changes over time).

Dempsey et al. [14] explored a meta-grammar approach to constant creation. The grammars that were used to create the potential solutions were evolved along with the solutions themselves. In this work, the effects of using different grammars were indirectly studied but no grammar analysis was conducted in detail and therefore, no grammar-construction guidelines were derived. Dempsey and colleagues found that the meta-grammar approach offered some advantages over other approaches on dynamic problems.

In this paper, we focus on the efficiency of the digit concatenation approach. It uses a grammar that includes the basic building blocks for number construction. For example, a grammar for synthesizing unsigned integer numbers is presented below.

Digit Concatenation Grammar

$$
\begin{aligned}
N &= \{< number >, < digitlist >, < digit >\} \\
T &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
S &= < number > \\
P &= \{ \\
< number > &\rightarrow \; < digitlist > \\
< digitlist > &\rightarrow \; < digit > \; | \; < digit >< digitlist > \\
< digit > &\rightarrow 0 \,|\, 1 \,|\, 2 \,|\, 3 \,|\, 4 \,|\, 5 \,|\, 6 \,|\, 7 \,|\, 8 \,|\, 9 \\
&\}
\end{aligned}
$$

This same approach, with small changes in the grammar, can be used for creating signed and floating-point numbers. Note that, in principle, it is possible to build numbers of any length. However, we will see later that this grammar induces a bias toward short-length numbers, making the pure digit concatenation approach inefficient when high-precision numbers are needed.

4 Number Length Bias in Digit Concatenation Grammars

The works described in Section 3 studied the ability of GE to find constants, the length of which (or of their components in the case of floating-point numbers) was short (between one and five digits) (see e.g., [13] and [7]). Interestingly, in their results one can notice that the error after several generations is still quite high in the case of "long" constants (those with at least 5-digit-long components). Since the focus of these works was on the relative performance obtained by GE when using different approaches for constant creation, this phenomenon remained largely unexplained.

Large errors when trying to build long numbers can be explained using simple concepts from the theory of stochastic context-free grammars [15], in which each production rule $r \in P$ has an associated probability $p(r)$ of being selected.

Consider a normal application of GE that uses a context-free grammar (CFG). We can consider the probability of selecting a production rule whose left-hand side nonterminal symbol is X, to be

$$p(r) = \frac{1}{|X|}, \tag{2}$$

where $|X|$ is the number of production rules that have X as their left-hand side symbol. Since we are using CFGs, the probability of a complete derivation is simply the product of the rule probabilities used and thus, the probability of generating a particular string (in our case, a number) is the sum of the probabilities of all possible derivations producing that particular string. For example, the probability of generating the number 5261 (in a GE style) from the digit concatenation grammar presented before is

$$\begin{array}{ll}
< number > \rightarrow\ < digitlist > & p_1 = 1.0 \\
< digitlist > \rightarrow\ < digit >< digitlist > & p_2 = 0.5 \\
\quad < digit > \rightarrow 5 & p_3 = 0.1 \\
< digitlist > \rightarrow\ < digit >< digitlist > & p_4 = 0.5 \\
\quad < digit > \rightarrow 2 & p_5 = 0.1 \\
< digitlist > \rightarrow\ < digit >< digitlist > & p_6 = 0.5 \\
\quad < digit > \rightarrow 6 & p_7 = 0.1 \\
< digitlist > \rightarrow\ < digit > & p_8 = 0.5 \\
\quad < digit > \rightarrow 1 & p_9 = 0.1
\end{array}$$

$$p(< number > \Rightarrow\ 5261) = p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9 = 6.25 \times 10^{-6}\,.$$

The probabilistic view presented above applies only when we generate strings at random, which is the case at initialization. The search algorithm behind GE will then try to adjust the population so as to increase the individuals' fitness (or reduce error). However, from a practical point of view, initial fitness is very important because it determines to a great extent the efficiency (i.e., the speed of convergence) of the approach.

In general, the probability of generating in the initial population a number of n digits (without instantiating any digit to a specific number) using the digit concatenation grammar is $1/2^n$. Clearly, the longer the length of the target number, the less likely it is to have a good approximation of it in the initial population.

5 Experiments

In our experiments, we measure the relative error $\delta_{\hat{x}}$ of the best solution $\hat{x}$ with respect to the target number value x. It is computed as follows

$$\delta_{\hat{x}} = \frac{|\hat{x} - x|}{|x|}, \tag{3}$$

where $x \neq 0$, which is always the case in our experiments.

The relative error measure is used as the fitness evaluation which is to be minimized. It guides a steady-state genetic algorithm which is used as search algorithm. At each trial, a different target number of length n is randomly generated in the range $[10^{n-1}, 10^n - 1]$. The parameters of the algorithm and the experiments are listed in Table 1.

Table 1. Parameters settings used in our experiments

Parameter	Value
Search algorithm	Steady state GA
Crossover operator	One point
Mutation operator	Bit-wise mutation
Population size	100
Number of generations	100
Probability of crossover	0.9
Probability of mutation	0.01
Population replacement strategy	100%
Wrapping events	No
Number of trials	100

The steady-state genetic algorithm with a 100% population replacement strategy effectively behaves as an algorithm with a $(\mu+\lambda)$ replacement strategy where the best individuals among parents and offspring are passed over to the next generation.

Figure 1 shows the relative error after 1000 fitness evaluations obtained by GE as a function of the length of the target number when using the classical digit concatenation grammar presented in Section 3.

The relative error grows with the target number length reaching a maximum value of 1.0. A maximum value of the relative error equal to 1.0 means that the best solutions found after 1000 fitness evaluations are insignificant (in terms of value) with respect to the long-length target numbers. It should be noted that the effectiveness of the approach is not under discussion. After 10000 fitness evaluations, the algorithm was capable of finding the target number irrespective of its length; however, it should be stressed that the focus of this study is on efficiency. In this respect, the results show clearly that building numerical values by using the classical digit concatenation grammar induces a strong bias toward short-length numbers.

The probability of having long-length numbers in the initial population decreases exponentially with the numbers' length. Ideally, this problem is solved by substituting the digit concatenation production rule by a rule with exactly the same number of digits as the target number. The rule in question is the following:

$$< number > \rightarrow \underbrace{< digit >< digit > \ldots < digit >}_{n \text{ digits}} .$$

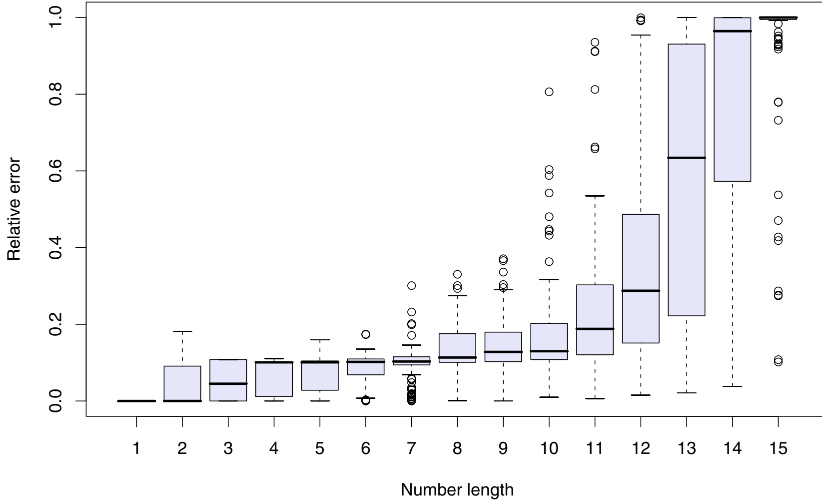


Fig. 1. Relative error as a function of the length of the target number. These results were obtained after 1000 fitness evaluations with the classical digit concatenation grammar.

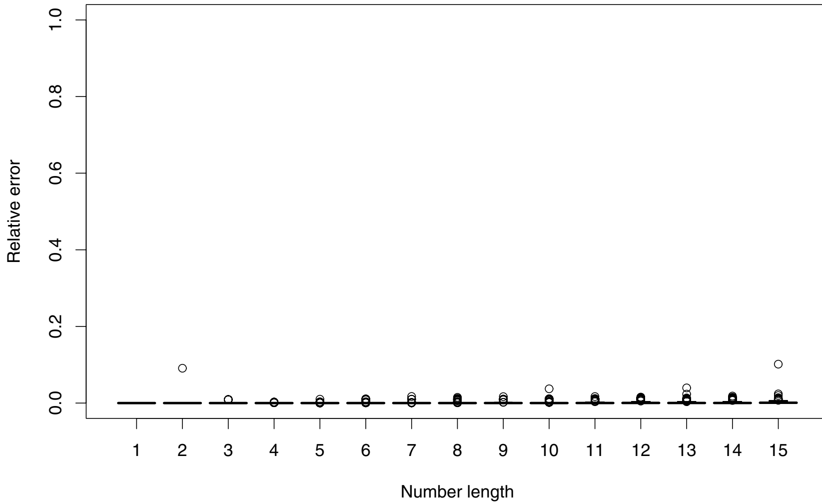


Fig. 2. Relative error as a function of the length of the target number. These results were obtained after 1000 fitness evaluations with an exact-length grammar. The grammar corresponds exactly with the length of the target number.

By using this rule, the problem is reduced to select the appropriate value for each of the digits of the target number. Effectively, short- and long-length numbers (up to a length of n digits) are generated with equal probability in this way. Figure 2 shows the relative error obtained by GE as a function of the length

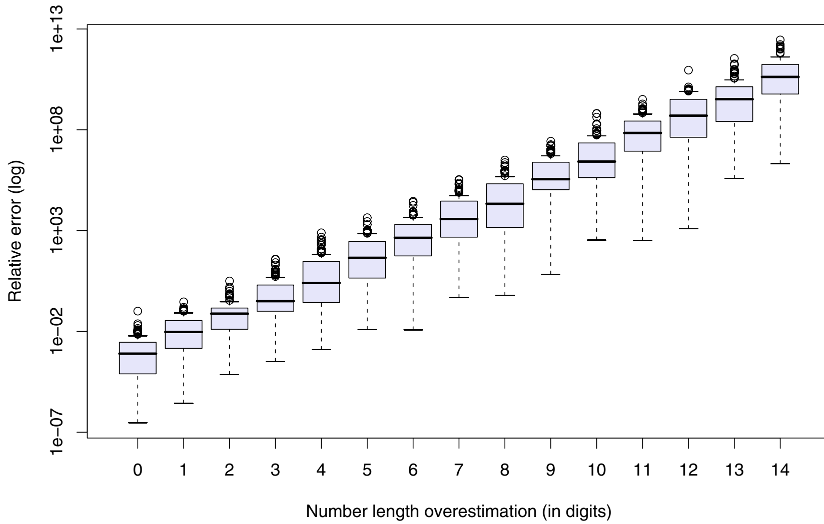


Fig. 3. Relative error obtained after 1000 fitness evaluations with an overestimated (in this case aimed at 15 digits) exact-length grammar. Note the logarithmic scale in the error axis.

of the target number using this exact-length grammar. Relative errors are small irrespective of the target number's length.

The solution just described suffers from one main drawback: It is necessary to estimate accurately the length of the target number before running the search process. If the length of the target number is greater than the estimation, the solution will fail miserably because it will not be possible to generate a number of the appropriate length. If the length of the target number is shorter than the estimate, the relative error will grow exponentially with the overestimation as shown in Figure 3.

An intermediate solution is thus proposed. It reduces considerably the bias toward short-length numbers and eliminates the need of estimating the target number's length beforehand. The production rules involved in the digit concatenation process are the following:

$$
\begin{aligned}
< number > &\rightarrow\ < digitlist > \\
< digitlist > &\rightarrow\ < digit > \mid < digit >< digitlist > \\
&\mid < digit >< digit > \mid < digit >< digit >< digitlist > \\
&\ldots \\
&\mid \underbrace{< digit >< digit > \ldots < digit >}_{k \text{ digits}} \\
&\mid \underbrace{< digit >< digit > \ldots < digit >}_{k \text{ digits}} < digitlist >,
\end{aligned}
$$

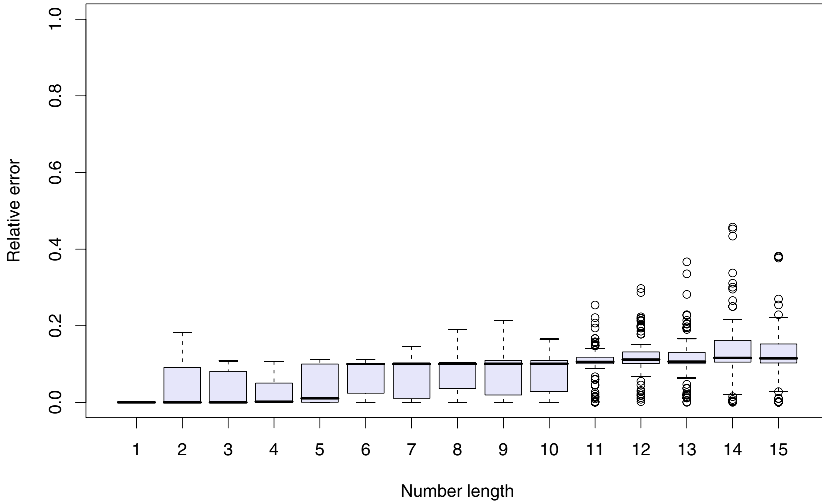


Fig. 4. Relative error as a function of the length of the target number obtained after 1000 fitness evaluations with a hybrid grammar

where $k <= n$ and n is the target number's length. Figure 4 shows the relative error obtained with a "hybrid" grammar in which $k = 5$.

In general, the obtained error is much higher than the one obtained with an exact-length grammar, but lower than the one obtained with a pure digit concatenation grammar. The parameter k determines the maximum size of the building blocks available to GE to produce numbers. It is expected that the greater k, the more uniform the distribution of numbers of different lengths in the initial population becomes.

6 Conclusions

Grammatical evolution (GE) is a relatively new evolutionary algorithm for the synthesis of programs or systems in any arbitrary representation language. This is possible thanks to a grammar-based search.

This paper highlights the importance of properly defining the grammar used by GE for the solution of a problem. One of the main strengths of GE is the possibility of biasing the search by means of a grammar; however, undesired biases can also be introduced. Although the focus here was on the synthesis of numerical values, the analysis based on stochastic context-free grammars can be applied to determine whether there is any undesired grammar-induced bias on other applications. A careful grammar design is needed in all cases.

This and previous studies have focused only on the synthesis of one numerical value at a time. This has been done because a detailed understanding of

the inner workings of grammatical evolution is necessary before embarking into more complicated studies. An investigation into the performance of GE on practically relevant application scenarios, in which several (not just one) numerical parameters are to be found, should be done in the future.

Acknowledgments

The author is grateful to Thomas Stützle, Mauro Birattari and the anonymous reviewers for their comments and suggestions to improve this paper. The author is funded by the Programme Alβan, the European Union Programme of High Level Scholarships for Latin America, scholarship No. E05D054889MX, and the *SWARMANOID* project funded by the Future and Emerging Technologies programme (IST-FET) of the European Commission (grant IST-022888).

References

1. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge (1992)
2. Koza, J.R.: Automatic synthesis of topologies and numerical parameters. In: Glover, F., Kochenberger, G.A. (eds.) Handbook of Metaheuristics, pp. 83–104. Kluwer Academic Publishers, Boston (2003)
3. Evett, M., Fernandez, T.: Numeric mutation improves the discovery of numeric constants in genetic programming. In: Koza, J.R., et al. (eds.) Genetic Programming 1998: Proceedings of the Third Annual Conference, pp. 66–71. Morgan Kaufmann, San Francisco (1998)
4. Topchy, A., Punch, W.F.: Faster genetic programming based on local gradient search of numeric leaf values. In: Spector, L., et al. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001), pp. 155–162. Morgan Kaufmann, San Francisco, CA, USA (2001)
5. Li, X., Zhou, C., Nelson, P.C., Tirpak, T.M.: Investigation of constant creation techniques in the context of gene expression programming. In: In Keijzer, M. (ed.) GECCO 2004. LNCS, vol. 3103, Springer, Heidelberg (2004) (Late Breaking Paper)
6. O'Neill, M., Ryan, C.: Grammatical Evolution. Evolutionary Automatic Programming in an Arbitrary Language. Kluwer Academic Publishers, Dordrecht (2003)
7. O'Neill, M., Dempsey, I., Brabazon, A., Ryan, C.: Analysis of a digit concatenation approach to constant creation. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E.P.K., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 173–182. Springer, Heidelberg (2003)
8. O'Neill, M., Ryan, C.: Grammatical evolution. IEEE Transactions on Evolutionary Computation 5(4), 349–358 (2001)
9. Brabazon, A., O'Neill, M.: Biologically Inspired Algorithms for Financial Modelling. Springer, Berlin (2006)
10. Cleary, R., O'Neill, M.: An Attribute Grammar Decoder for the 01 MultiConstrained Knapsack Problem. In: Raidl, G.R., Gottlieb, J. (eds.) EvoCOP 2005. LNCS, vol. 3448, pp. 34–45. Springer, Heidelberg (2005)
11. Tsoulos, I.G., Gavrilis, D., Glavas, E.: Neural network construction using grammatical evolution. In: Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, Piscataway, NJ, USA, pp. 827–831. IEEE Press, Los Alamitos (2005)

12. Dempsey, I., O'Neill, M., Brabazon, A.: Grammatical constant creation. In: Deb, K., et al. (eds.) GECCO 2004. LNCS, vol. 3103, pp. 447–458. Springer, Heidelberg (2004)
13. Dempsey, I., O'Neill, M., Brabazon, A.: Constant creation in grammatical evolution. International Journal of Innovative Computing and Applications 1(1), 23–38 (2007)
14. Dempsey, I., O'Neill, M., Brabazon, A.: meta-Grammar Constant Creation with Grammatical Evolution by Grammatical Evolution. In: GECCO 2005: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, pp. 1665–1671. ACM Press, New York (2005)
15. Manning, C., Schütze, H.: Foundations of Statistical Natural Language Processing. MIT Press, Cambridge (1999)

Cooperative Problem Decomposition in Pareto Competitive Classifier Models of Coevolution

Andrew R. McIntyre and Malcolm I. Heywood

Faculty of Computer Science, Dalhousie University
6050 University Avenue. Halifax. NS. Canada
{armcnty,mheywood}@cs.dal.ca

Abstract. Pareto competitive models of coevolution have the potential to provide a number of distinct advantages over the canonical approach to training under the Genetic Programming (GP) classifier domain. Recent work has specifically focused on the reformulation of training as a two-population competition, that is learners versus training exemplars. Such a scheme affords, for example, the capacity to decouple the fitness evaluation overhead from the data set size through sub sampling while naturally encouraging 'teams' or composite solutions as opposed to solutions based on a single individual alone. One outstanding question with respect to the latter characteristic is with regards to the nature of the team (archive) behavior in terms of pattern coverage. That is to say, which models are used when, and what are the implications for solution modularity as it relates, for example, to the assignment of exemplars to solution participants. The current work investigates two Pareto competitive approaches to classification under GP, with one configured to employ an explicitly cooperative multi-objective cost function based and the other employing the classical (error-based) cost function. We empirically demonstrate a critical distinction between the two with regards to problem decomposition, with the capacity to provide a decomposition into unique behaviors being much more prevalent when co-operative mechanisms are explicitly supported.

1 Introduction

Pareto competitive coevolution has begun to make the transition from the genetic algorithm (GA) to genetic programming (GP) domain. The basic Pareto competitive coevolutionary model of relevance to the classification domain encourages evolution to take place between one or more populations such that one set of individuals represent a set of test cases (training exemplar subset) and the other a set of learners. A Pareto ranking model is established in which the test cases are rewarded for their ability to distinguish between the learners, and the learners are rewarded for their ability to remain non-dominated (with respect to the set of test conditions) [2], [10], [1]. The end result is that two Pareto fronts are built, one in which a set of non-dominated test points exist (corresponding to the minimal set of tests necessary to distinguish between the current set of

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 289–300, 2008.
© Springer-Verlag Berlin Heidelberg 2008

learners) and the second details the corresponding set of non-dominated learners. The advantage that such a model provides is twofold. Fitness evaluation need no longer be conducted over the entire set of training exemplars, and a formal process has been established for identifying the 'best' training points and learners. Moreover, it is also possible to interpret convergence in terms of the behavior of the learner Pareto front, thus addressing the issue of domain independent stop criterion[1]. To date, however, the GA bias to this research has emphasized the issues of diversity maintenance, monotonic progress, and niching within the learner population.

In a GP setting the same issues exist, enabling some of the solutions to be carried over from a GA context. In particular, GA Pareto competitive models have been relatively successful in establishing a basis for coevolving two population models under the GP classification domain [5], [6], [12], [8]. In this work we will examine one of the basic problems unique to the GP classification domain under a Pareto competitive model, and illustrate how it can be dealt with by introducing a cooperative model of fitness assignment in addition to the Pareto competitive coevolutionary model.

Specifically, under the classification domain the Pareto competitive model results in a front of solutions for both learner and (test) point populations. This means that the solution will no longer be in the form of a single classifier, but in terms of a set of classifiers (those in the Pareto front). The basic issue at stake here is how to determine which model to use when. The problem does not appear in a GA setting for the most part because individuals take the form of a co-ordinate in a multi-dimensional space, thus application of an appropriate distance metric is sufficient to resolve which individual to apply when. Moreover, the GA domain may also introduce diversity mechanisms based on Euclidean distance metrics to encourage desirable 'coverage' properties in the Pareto front itself. Finally, although GA domains might utilize a Pareto method to evolve a front of solutions, they often assume that only one is chosen for deployment by the user.

Conversely, under a GP classification domain we demonstrate in this work that a lot depends on how the fitness function and associated wrapper operator are designed. In particular, the most straightforward approach might assume a binary (hit) based model on account of the ease with which the associated Pareto dominance test might be made. We show in this work that this results in a weak learner type of association between learners and exemplars, with significant overlap between the exemplars and responding learners. Conversely, by assuming a cooperative multi-objective model to fitness evaluation (in addition to the Pareto competitive model of evolution) we are able to establish an explicit decomposition of exemplars to learners. As such, the post training assignment of learners to exemplars merely takes the form of utilizing the learner providing maximum class membership.

In the following, models for Pareto competitive coevolutionary and Pareto cooperative-competitive models of GP coevolution are introduced, Section 2.

[1] Originally proposed under the GA paradigm [4], and reapplied under GP [8].

In doing so, we contrast the utility of local and global wrapper operators, and make the case for establishing reward mechanisms that explicitly encourage cooperative behaviors. Both algorithms provide solutions in the form of a 'front' of solutions. Section 3 investigates the nature of the interaction between individuals within the front under three multi-class classification problems. The effectiveness of the cooperative–coevolutionary model is now clear, with solutions typically taking the form of a clear behavioral decomposition of the problem domain. Conversely, the competitive–descriminator based model typically results in solutions in which a complex mixture of classifiers takes place, without any improvement over the classification accuracy of the former model.

2 Pareto Competitive and Pareto Cooperative–Competitive Coevolution

In order to illustrate the aforementioned property of competitive coevolutionary GP classifiers, we compare the operation of two recent frameworks that utilize a Pareto based model of interaction between points (exemplars) and learners (classifiers): the Pareto-coevolutionary Genetic Programming Classifier [5], and Competitive Multi-objective Grammatical Evolution [8], [7]. For consistency both are implemented in terms of a canonical model of Grammatical Evolution (GE) [11], although both are entirely independent of the model of evolution on which they are based. Hereafter we refer to them as PGEC and CMGE respectively. In the following we establish the principle differences between the two models, and refer the reader to the original works for the detailed algorithmic descriptions.

The basic features of the CMGE classifier are summarized as follows relative to the pseudo code listing provided in Algorithm 1.

1. Identification of the subset of exemplars over which individual (learner) evaluation will take place (steps 2(a), 2(b));
2. Identification of the local membership function (wrapper operator) for each individual, relative to the associated *gpOut* distribution (steps 2(c)ii.A to D);
3. Fitness evaluation of individuals relative to the learning objectives under a multiobjective methodology (lines 2(c)ii.E to G);
4. Identification and archiving of the most valuable individual classifiers and exemplars (steps 2(d));
5. Class-wise assessment of early stopping criteria (steps 2(e)).

Conversely, the PGEC model is limited to: steps 2(a) and (b), define the content of the learner and point archives, after which the outcome vector for each individual is established, and then step 2(d) is performed, that is the Pareto assessment for establishing archive content.

2.1 Competitive Multi-objective Grammatical Evolution

The standard initialization process of step 1, Algorithm 1 stochastically creates GP population members (learners) and prepares the relevant data structures,

including archives for both learners and exemplars (data points or simply, points). A while loop (step 2) encloses the main sections of the algorithm, ensuring that the training of GP is repeated until stopping conditions are met (as evaluated at the end of each iteration in step 2(e)). Steps 2(a) and 2(b) set up the training subset at each iteration ensuring a balanced view of data, thus enabling robustness against problems having unbalanced class distributions.

Algorithm 1. *High-level Pareto Coevolution.*

1. *Initialize Learner Population (LP);*
2. *WHILE ! (Stop criteria)*
   (a) *Point Population (PP) := random balanced sample of training partition;*
   (b) *Training Subset (TS) := PP concatenated with Point Archive contents (PA);*
   (c) *FOR i := 1 to sizeof(LP)*
      i. *Apply variation operators to Produce Children (C)*
      ii. *FOR j := 1 to sizeof(C)*
         A. *Establish phenotype of individual C[j];*
         B. *Map TS to 1-d number line 'gpOut' of C[j];*
         C. *Cluster gpOut of C[j];*
         D. *Parameterize Gaussian Local Membership Function (LMF) of child C[j];*
         E. *Evaluate C[j] with respect to:*
         SSE, Overlap wrt. Learner Archive (LA), Parsimony.
         F. *Rank C[j] with respect to LP and assign fitness;*
         G. *Replacement (insert C[j] into LP);*
   (d) *Archive PP, LP members based on outcomes (according to IPCA)*
      i. *Points in PP enter PA if they provide a new distinction;*
      ii. *Learners in LP enter LA if they are non-dominated wrt. LA;*
   (e) *Evaluate Stop Criteria (method of Rank Histograms);*
3. *Class-wise LA denote solution: Build appropriate weighting scheme;*

Line 2(c) of Algorithm 1 begins the cooperative coevolution training loop which employs an Evolutionary Multi-objective Optimization (EMO) model loosely based on that of [4] to train GP. On each pass of the loop, selection and variation operators are applied to the GP population and children are produced (line 2(c)i). Next, individuals are decoded to their respective phenotype (line 2(c)ii.A) and the current selection of exemplars are mapped to the *gpOut* axis. We now require a mechanism to identify the local membership function (wrapper operator) neighborhood without resorting to inappropriate or arbitrary predefinitions of regions along the *gpOut* axis. In order to achieve this goal we assume that the neighborhoods of most relevance are those having the highest density, a requirement satisfied by a clustering algorithm (step 2(c)ii.C). The clustering algorithm returns the location of the mid point associated with the 'most dense' set of points and exemplars associated with this cluster, *M*. The

process itself is independent of class label. We now have the properties for the local membership function (LMF) defined in terms of a Gaussian with mean, μ, and variance, σ (line 2(c)ii.D).

A fitness function is now applied to the subset of points of the neighborhood, M (line 2(c)ii.E, Algorithm 1). The objectives are designed to encourage: least ambiguity in cluster membership, non overlapping behavior of the exemplars mapped to different individuals, maximization of the number of in-class exemplars mapped to an individual, and simplicity of the GP mapping. Note that, in common with the findings of other EMO research, we establish a set of objectives that have a degree of implicit 'tension' between them. In doing so we are able to encourage mappings that reduce the likelihood of degenerate solutions. Moreover, in order to measure these objectives, the mapping is assigned a class, where this is assumed to correspond to the class of the point at the center of the LMF. In taking this route we avoid making any assumptions regarding which individuals are mapping which classes, and effectively let individuals compete for the right to map exemplars. The inner loop is completed by returning to the generic PCGA EMO algorithm of [4] in order to complete the Pareto ranking and replacement policy (lines 2(c)ii.F and G respectively). The significance of the Pareto ranking and ensuing fitness assignment is that selection operators proportionately favor individuals of higher fitness (lower ranking) over those having lower fitness (higher ranking). This tends to encourage the GP algorithm to more frequently sample material corresponding to individuals that lie closer to the Pareto front, with the goal of establishing improvements in the objectives. The PCGA model also provides the concept of rank histograms, which essentially summarizes the content of the population (in objective space) in terms of the Pareto ranks, so that content can be readily compared between training epochs. When calculated for each class, this provides the basis for the detection of class wise early stopping, line 2(e).

The inner loop defined by line 2(c) denotes the cooperative EMO model. This portion of the main loop is performed in combination with the Pareto competitive model[2] for the purpose of adapting learner and test point archives as memories at line 2(d). That is to say, the competitive coevolution model's evaluation is conducted over the contents of the subset of training exemplars, TS (step 2(b)), dynamically identified by a competitive co-operative model for archiving the most discriminatory test points (step 2(d)i) and non dominated learners (step 2(d)ii), both from the perspective of a Pareto front. The competitive model thus plays a primarily archival role, acting as a memory for the cooperative model. The archive entry criteria are evaluated in terms of GP classification 'outcomes' which are directly related to the LMF definition and it's associated performance on the training set i.e., the outcome vector is takes on real values as opposed to the binary case of IPCA.

Deployment of the classifier (step 3) takes the form of copying the contents of the learner archives and assignment of weights to each on the basis of the

[2] A variant of de Jong's IPCA algorithm [12], although any of this class of algorithm would be appropriate.

training data. A winner-take-all policy with respect to LMF response determines the assignment of class labels among the team individuals.

2.2 Discussion

The critical differences between the PGEC and CMGE models are the use of EMO fitness evaluation in which cooperative behavior is explicitly sought in the mapping between exemplars and class membership, step 2(c), Algorithm 1. PGEC instead relies on the standard sigmoid based global wrapper operator for the purpose of mapping *gpOut* to class labels. This also implies that PGEC is a binary classifier, requiring multiple runs to evolve classifiers for each class, whereas CMGE provides classifiers for all classes from a single run. The interface to IPCA, line 2(d) remains unchanged i.e., the outcome vector. As such the principle mechanism for encouraging problem decomposition is the competitive model of Pareto dominance, as expressed between points and learners. Given that there is no explicitly cooperative mechanism for establishing population diversity, we maintain that this will generally result in PGEC producing classifiers with overlapping behaviors. That is to say, learners need only differ in one outcome in order to satisfy the Pareto dominance criterion and appear in the learner archive.

Unlike IPCA, both CMGE and PGEC make use of heuristics to enforce finite archive sizes for point and learner archives, PA and LA. In the case of the point archive, both PGEC and CMGE replace points once the archive limit is reached using an Euclidean distance metric in which the nearest current point is replaced [5]. In the case of the learner archive both PGEC and CMGE replace the individual currently within the archive with largest overall error (as estimated against the current training subset, TS).

A second difference resulting from the two models appears in the post training voting mechanism, step 3, Algorithm 1. PGEC relies on a majority policy, where this is designed to make use of the expected overlap in learner archive classifier behavior. Conversely, CMGE learners are expected to be unique, thus a winner takes all policy is assumed. In the case of PGEC, the merit of assuming a particular policy is expected to be more significant, as the degree of interaction between learners is likely to be data set specific. Conversely, under CMGE a winner takes all policy is a natural consequence of the explicitly cooperative model of evolution, reinforced by the action of the Gaussian local membership function.

3 Results

The PGEC and CMGE models are implemented using a common grammar and set of evolutionary parameters, Table 1. The grammar is capable of specifying zero argument (exemplar features), single argument (cosine, square root, natural log, exponential), and double argument (plus, minus, multiply, divide) operators. Variation operators take the form of one point crossover and mutation (PXO and

PM respectively) and their corresponding context aware variants (PCXO and PCM respectively) [3]. Classifiers are implemented as a 'parallel model' in which a 'k' class problem implies that 'k' learner archives are evolved. A larger study, [7], conducted an evaluation over nine additional multi-class data sets taken from the UCI repository [9]. In this work, we focus on three interesting cases that characterized the behavior of all nine cases: Iris (IRIS), Boston Housing (BOST), and Contraceptive (CONT). Table 2 characterizes the basic properties of the data sets as deployed in this study. All three data sets are three class problems, and results employ ten fold cross validation with fifty runs per fold. The only pre-processing performed involved removing duplicate and incomplete exemplars from the original data set.

Table 1. GE Parameterization

Parameter	Value	Parameter	Value
Max Generation	500	Learner Pop Size	50
Max Codon	256	Learner Archive Size	30
Max Codon Trans.	4,096	Point Pop Size	30
–	–	Point Archive Size	30
PXO (PCXO)	0.5 (0.9)	PM (PCM)	0.01 (0.9)

Table 2. Data Set Characterization

Data set	num. Exemplars	num. Features	Class distribution (percent)
IRIS	147	3	33–33–33
BOST	506	12	33–33–33
CONT	1,425	8	43–22–35

3.1 Evaluating Intra-class Voting Behavior

In order to investigate the effectiveness of the cooperative mechanism in CMGE versus the PGEC model, a metric for intra-class voting behavior is derived. In particular we wish to measure the degree to which learners comprising the Pareto front form constructive interactions, that is decompose the problem into non overlapping associations between learners and exemplars. This is interpreted in terms of the strength of the class membership operators, Gaussian and Sigmoid (local and global) for CMGE and PGEC respectively, relative to the median performance of the set of individuals constituting the 'team' of the same class. That is to say, given a winning classifier (the individual with maximum membership on an exemplar) we measure the difference in membership of the winner relative to the median membership of other classifiers associated with the same class. Differences would be distributed over the unit interval, and results in the metric characterizing performance in terms of three generic outcomes,

- Differences tending towards zero: indicates that there is little difference between membership of winning classifier and the median classifier performance. Needless to say, this could be associated with the majority of individuals labeling an in class exemplar correctly or incorrectly;
- Differences tending towards the mid point (0.5): indicates an individual with strong winning class membership, but with the majority of 'runner up' in-class individuals responding with a 'fifty percent' membership. Thus, the winning individual had a membership in the interval [0.5, 1], with the majority of the remaining intra-class classifiers responding with membership in the interval [0.25, 0.75]. Such behavior is considered undesirable as it is no longer possible to establish a clear difference between individuals labeling in class behaviors and those associated with out of class behavior.
- Differences tending towards unity: implies that the class winner responds with a class membership tending towards unity, whereas the majority of the other individuals respond with low class membership. Naturally, this implies a strong uniqueness in the classifier to exemplar decomposition.

The following comparison will first establish the baseline performance of each model in terms of detection rate, false positive rate and the number of participating models. This establishes that nothing is lost by assuming a model that enforces cooperative problem decomposition. The second evaluation characterizes the nature of the intra-class decomposition.

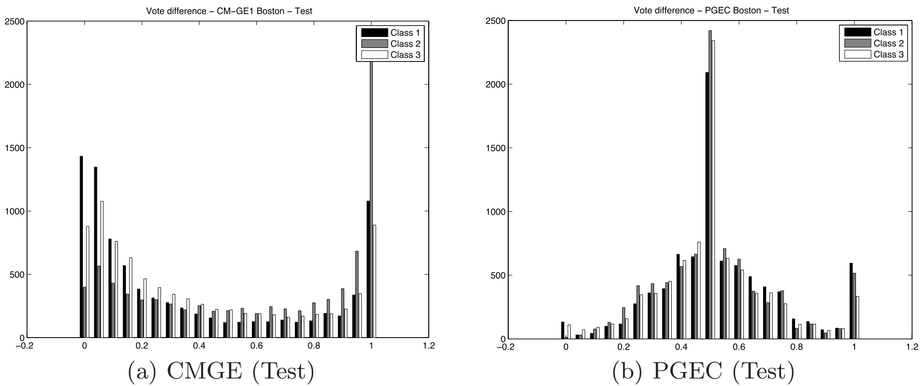
Complexity and Classification Performance. Table 3 establishes the number of classifiers participating and ensuing detection / false positive rates over the test partition in terms of the class-wise median. Both models clearly utilize multiple classifiers per class. It is also clear that the CMGE model provides a more reliable classifier, with the significantly higher per class detection rates more than out weighing any increases in false positive rate. Moreover, the PGEC model was uncompetitive on both the 'balanced' Iris data set as well as the larger unbalanced data sets. Also of significance is that this is typically achieved while CMGE utilizes the entire contents of the learner archive. Thus only on class one of IRIS did CMGE employ a much lower count of classifiers than that in the remaining cases (all of which tend to the archive limit of thirty).

Intra-class Coverage. The above section established that the explicitly cooperative model of GMGE is able to build on the competitive coevolutionary paradigm shared by both models. In this section we characterize the form of the decomposition using the aforementioned coverage metric. Specifically, we build histograms of the CMGE and PGEC intra-class coverage over the test partition (no significant differences appearing between training and test histograms). Figures 1, 2 and 3 summarizing the basic behaviors on the Boston Housing, Iris and Contraceptive data sets respectively. In the case of the Boston Housing data set, CMGE results in a bimodal distribution in which there is either a considerable differentiation between winning classifier and the remainder of the same class

Table 3. Median Test Set Performance

Classifiers per Class			
Data set	IRIS	BOST	CONT
Class	1–2–3	1–2–3	1–2–3
CMGE	3–30–30	30–30–30	29–30–30
PGEC	1–5–4	10–14.5–10	14–15–21
Detection Rate			
CMGE	100–100–100	87.5–35.3–82.4	73.8–31.2–24
PGEC	0–100–40	17.6–52.9–6.2	18–11.1–32.7
False Positive Rate			
CMGE	0–0–0	22.9–9.4–15.2	53.7–14.5–16.3
PGEC	0–50–0	6.1–44.1–2.9	11.1–9.4–28

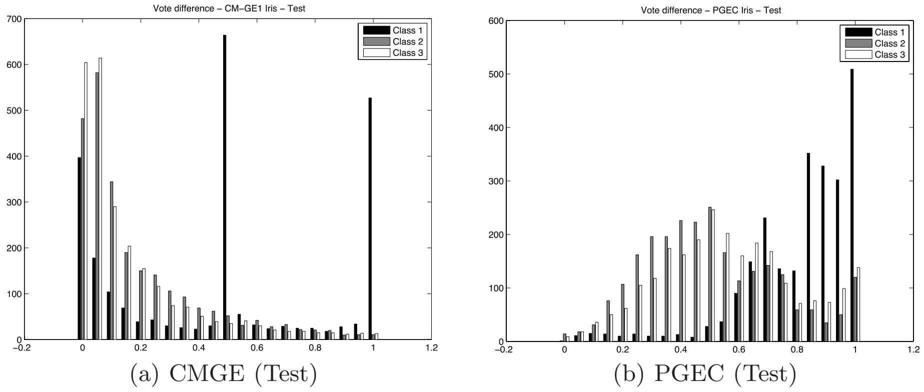
classifiers (the right peek at unity), or the majority of the classifiers have a similar class membership behavior (the left peek at zero). Moreover, class 2 appears to result in classifiers demonstrating most behavioral uniqueness, whereas classes 1 and 3 result in behavior distributed equally at the two peeks. PGEC on the other hand demonstrates a strong preference for multiple individuals responding at an intermediate level of class membership (i.e., the peak at 0.5). As such it is not possible to establish that the majority of in-class individuals respond with a strong in-class preference or a strong differentiation between in and out of class behavior, Figure 1(b).



(a) CMGE (Test) (b) PGEC (Test)

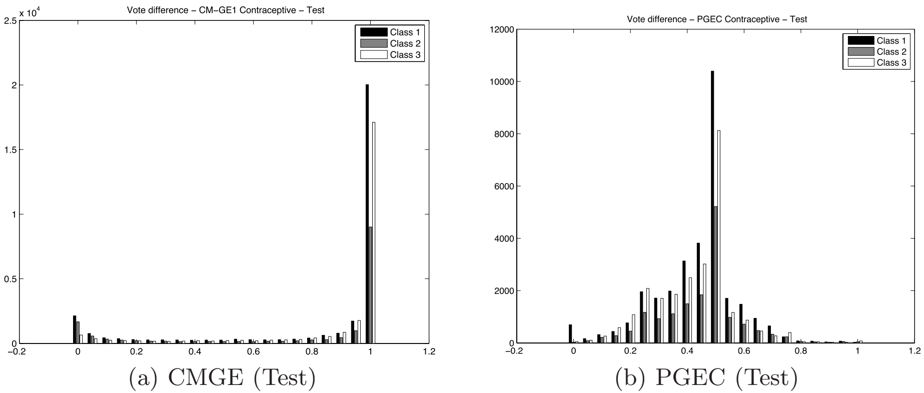
Fig. 1. Team behavior: Boston data set

Under the Iris data set, CMGE demonstrates two distinct distributions. In the case of the linearly separable class (one) a distribution similar to that for the Boston Housing data set is returned i.e., strong similarity or strong differentiation. The single peak at the mid point is, in this case, produced as an artifact of an equal number of in-class classifiers returning both maximum (1) and minimum (0) differences. On the two non-linearly separable classes the intermixing



(a) CMGE (Test) (b) PGEC (Test)

Fig. 2. Team behavior: Iris data set



(a) CMGE (Test) (b) PGEC (Test)

Fig. 3. Team behavior: Contraceptive data set

of the class boundary results in a bias towards a higher similarity in behavior between in-class classifier membership. Given the strong classification performance of the model as a whole, Table 3, this implies that multiple classifiers are involved in supplying a correct label. The PGEC model also produces two distinct distributions under the Iris data set. However, although the linearly separable class does result in a desirable peak at unity, there are also secondary distributions in the interval 0.8 to 1 and around 0.7. The two non linearly separable classes (two and three) appear to rely on a complex intermixing of class votes with no clear differentiation between winner and other in-class classifiers. In short PGEC appears to find it difficult to encourage individual classifiers to take a definite policy regarding the subset of exemplars on which they will respond. This is reinforced by the poor overall classification results, Table 3.

Finally, the Contraceptive data set, Figure 3, is representative of the most difficult problem domain considered and in the case of CMGE results in a clear

emphasis towards classifiers that respond to very different subsets of exemplars (high count of large differences). Conversely the PGEC model is generally unable to establish a clear differentiation in class membership values, with most of the distribution again sitting in the mid region of the histogram.

4 Conclusion

Coevolutionary models of classification using GP are beginning to appear in which the test point competitive coevolutionary Pareto models developed under a GA setting by Watson, Ficici, de Jong, and Pollock frequently serve as the starting point. The model provides many useful properties, not least that the inner loop of GP is now decoupled from the size of the original training data set. In this work we emphasize that there are also several GP and classification domain specific problems that were not especially relevant in the original GA domain. In particular, just because the solution (typically) takes the form of a set of classifiers (the contents of the learner archive or Pareto front), this is not sufficient to encourage distinct behaviors between the learners themselves. Related to this property is the need to introduce a mechanism for establishing post-training class labels from the ensuing classifiers. In this work we revisit the original Pareto competitive classification model of Lemczyk [5] in which a global membership function and majority voting are used to establish class labels, and compare with a local membership function in which members of the Pareto front are required to explicitly cooperate under a local wrapper operator [8],[7]. In order to investigate this property a 'coverage' metric is introduced to establish the degree of differentiation between 'winning' class behavior and the median performance of the remaining classifiers. The resulting evaluation clearly demonstrates that the competitive-coevolutionary CMGE model is able to associate specific exemplar subsets with specific classifiers, whereas PGEC is unable to provide a clear separation between classifier behaviors. Moreover, this is achieved without compromising the performance of the ensuing classifiers.

Future work will revisit the representation used within the context of the point population. In particular the GP domain typically employs a GA population for the points in which exemplars are directly represented by indices. The basic problem with this is that although directly supporting the competitive coevolutionary model, there is no natural mechanism for establishing context on which a crossover operator could operate. Thus, to date the most effective model appears to simply re-establish the point population at each generation using uniform selection with a class balance enforcing heuristic, whereas finding a representation that permits variation operator context might provide a more elegant solution.

Acknowledgements

The authors gratefullly acknowledge the support of PRECARN, NSERC Discovery, MITACS, and CFI New Opportunities programs; and industrial funding through the Telecom Applications Research Alliance (Canada), and SwissCom Innovations AG (Switzerland).

References

1. de Jong, E.D., Pollack, J.B.: Ideal evaluation from coevolution. Evolutionary Computation 12(2), 159–192 (2004)
2. Ficici, S.G., Pollack, J.B.: Pareto optimality in coevolutionary learning. In: European Conference on Artificial Life, pp. 286–297 (2001)
3. Harper, R., Blair, A.: A structure preserving crossover in Grammatical Evolution. In: IEEE Congress on Evolutionary Computation, vol. 3, pp. 2537–2544 (2005)
4. Kumar, R., Rockett, P.: Improved sampling of the pareto-front in multi-objective genetic optimizations by steady-state evolution. Evolutionary Computation 10(3), 283–314 (2002)
5. Lemczyk, M., Heywood, M.I.: Training binary GP classifiers efficiently: a pareto-coevolutionary approach. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 229–240. Springer, Heidelberg (2007)
6. Lichodzijewski, P., Heywood, M.I.: Pareto-coevolutionary Genetic Programming for problem decomposition in multi-class classification. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), vol. 1, pp. 464–471 (2007)
7. McIntyre,A.R.: Novelty Detection + Coevolution = Automatic Problem Decomposition: A Framework for Scalable Genetic Programming Classifiers. PhD thesis, Faculty of Computer Science, Dalhousie University (2007), http://www.cs.dal.ca/∼mheywood/Thesis
8. McIntyre, A.R., Heywood, M.I.: Multi-objective competitive coevolution for efficient GP classifier problem decomposition. In: Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 1930–1937 (2007)
9. Newman, D.J., Hettich, S., Blake, C.L., Merz, C.J.: UCI repository of machine learning databases (1998), http://www.ics.uci.edu/~mlearn/mlrepository.html
10. Noble, J., Watson, R.: Pareto coevolution: Using performance against coevolved opponents in a game as dimensions for pareto selection. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), pp. 493–500 (2001)
11. O'Neill, M., Ryan, C.: Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language, May 2003. Springer, Heidelberg (2003)
12. Yo, T.S., de Jong, E.D.: A comparison of evaluation methods in coevolution. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO). SIGEVO, vol. 1, pp. 479–486 (2007)

Integrating Categorical Variables with Multiobjective Genetic Programming for Classifier Construction

Khaled Badran and Peter Rockett

Laboratory for Information and Vision Engineering,
Department of Electronic and Electrical Engineering,
The University of Sheffield,
Mappin Street,
Sheffield S1 3JD, UK
khaled.badran, p.rockett@shef.ac.uk

Abstract. Genetic programming (GP) has proved successful at evolving pattern classifiers and although the paradigm lends itself easily to continuous pattern attributes, incorporating categorical attributes is little studied. Here we construct two synthetic datasets specifically to investigate the use of categorical attributes in GP and consider two possible approaches: indicator variables and integer mapping. We conclude that for ordered attributes, integer mapping yields the lowest errors. For purely nominal attributes, indicator variables give the best misclassification errors.

1 Introduction

Genetic programming (GP) has been used successfully to evolve classifiers [5, 7, 9, 11, 13]. A GP tree is well-suited to performing pre-processing tasks and feature extraction on *continuous* pattern attributes, however there are many pattern recognition tasks of practical interest where the pattern attributes are either purely categorical or mixed continuous/categorical. How to handle categorical or mixed attributes in classification is a general problem. Decision trees and Bayesian belief networks are inherently based on categorical attributes and continuous attributes generally have to be discretised for use with these classifiers. Classifiers such as support vector machines, neural networks and linear discriminants, on the other hand, are fundamentally continuous paradigms and their use with categorical variables usually requires special pre-processing – see, for example, [12].

(In fact, there are two types of categorical variables: *Ordinal* and *nominal*. Ordinal attributes have a clear ordering, for example, $Height > 1.5m$, $Height > 1.6m$, etc. Nominal values on the other hand, have no such order, for example: *Red*, *Blue*, *Green*, etc.)

Using GP-derived classifiers with categorical variables is of significant interest given the promise shown by the approach on continuous variable classification

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 301–311, 2008.
© Springer-Verlag Berlin Heidelberg 2008

problems but as far as we are aware, there has been only one report studying how to integrate categorical variables into GP classifiers [10]. Loveard & Ciesielski investigated four strategies for using categorical attributes:

1. Mapping each category to an integer value. For example, an attribute value of *Red* returns 0, a value of *Green* returns 1, an attribute value of *Blue* returns 2, and so on. This mapping generally corresponds to some *ordering* of the data and makes the implicit assumption that it is possible to define a distance metric in the attribute space.
2. Using indicator variables, that is, defining a node type for every value of every attribute. So for the case of three attribute values, *Red*, *Green* and *Blue*, we have three nodes. The first type returns the numerical value of unity if the value of the pattern attribute is *Red*, otherwise it returns zero, and so on for nodes which return unity when the attribute value is *Blue* and zero otherwise, and one which returns unity when the attribute is *Green* but zero otherwise. Indicator variables have a long history in data analysis.
3. A multi-way branch node which, for example, executes the first sub-tree if the attribute value is *Red*, the second sub-tree if the attribute value is *Green* and the third sub-tree if the attribute value is *Blue*.
4. A series of 'if-then-else' nodes, one per possible attribute value. So there is one node where the left sub-tree is executed if the attribute value is *Red* but the right sub-tree is executed if the attribute value is not *Red*. And so on for all other possible attribute values.

See [10] for further details. Unfortunately, the report by Loveard & Ciesielski was brief (4 pages) and thus contained too few experimental details to allow replication of their results with any degree of certainty. Further, their study was *phenomenological* in that it examined real datasets but could only conclude that the best strategy "varied from one (real) dataset to another". The principal motivation of the current work has been to gain a greater understanding of the problem and to produce more concrete guidelines on handling categorical variables than were yielded by [10].

Of the possible strategies enumerated above, the third and fourth require the definition of new node types for every problem and for categorical attributes which can take a large number of possible values, the size of the trees will increase proportionately, increasing the size of the search space. Therefore, in this paper we consider only the first two strategies: Integer mapping and indicator variables. On the basis of the experiments reported here we reach firm conclusions on the range of applicability of each strategy.

In the following section we outline the GP environment we have used to conduct our experiments. In Sections 3 and 4 we describe two specially-constructed synthetic datasets and associated experiments designed to elucidate the domains of applicability of the two strategies. In Section 5, we report results on a range of both categorical and mixed real datasets and offer overall conclusions in Section 6.

2 Multiobjective Genetic Programming Environment

The multiobjective GP environment used in this work minimises the two objectives of: Tree node count and misclassification error over the training set. The tree node count objective imposes a parsimony pressure which has been shown to be highly effective in preventing bloat [3]. We use the Pareto multiobjective framework described by Fonseca & Fleming [4] and the evolutionary strategy was a steady-state $(\mu + 2)$ algorithm [8] with the worst-ranked two individuals being discarded after every breeding cycle.

For all the experiments reported here, we have used a fixed population size of 100 which was initialised with half being randomly-generated trees of depth 7, and the other half randomly generated trees of randomly chosen depths in the range $[1 \dots 7]$. We have used the depth-fair crossover operator due to Ito et al. [6] together with point mutation which gives every sub-tree an equal probability of being chosen for mutation and thereby replaced with a randomly-generated tree of randomly selected depth $\in [2 \dots 4]$.

The function set comprised: $+$, $-$, $\times$ and protected division. We used three possible terminal nodes: continuous attributes, terminals which map each category to an integer value and terminals which implement indicator variables. Having projected each pattern vector into a 1D decision space with the GP-derived tree, we determine an optimal decision threshold within the evolutionary loop using Golden Section search.

For the synthetic data described in Sections 3 and 4 we have used a fixed number of 20,000 tree evaluations – at this point, all further improvement in the population had long since ceased. For the real datasets described in Section 5, we continued each GP run until further improvements in the population had long ceased; the exact numbers of tree evaluations vary from problem-to-problem.

3 Synthetic Ordinal Dataset

3.1 Dataset

In the first series of experiments we have constructed 2-class synthetic data using Gaussian distributions, where the numbers of dimensions, $D \in [2, 4, 6]$. One class is assigned a zero mean vector $(\boldsymbol{\mu_1})$ while the mean vector of the other class, $\boldsymbol{\mu_2}$ is given by:

$$\mu_{2,i} = \frac{\alpha}{\sqrt{D}} \quad \text{for} \quad i \in [1 \dots D]$$

where scaling the vector elements by $\sqrt{D}$ ensures a fixed Mahalanobis distance between the class means and hence the Bayes error is independent of D. Thus for convenience, all misclassification errors are on approximately the same scale. α was adjusted to give a convenient misclassification error of around 10-15% and held constant thereafter at a value of 0.8. The covariance matrices of both classes were taken as the identity matrix.

We generated 10 training data per class per dimension and 100 test data per dimension per class. To convert this continuous data into ordinal data, we discretised (binned) each attribute into varying numbers of bins, N_B, where the width of each bin was given by:

$$\frac{3 \times \mu_{2,i}}{N_B}$$

and the two outermost (guard) bins extend to $+\infty$ and $-\infty$, respectively.

Each continuous pattern attribute was replaced with the index of the bin into which it fell, starting with an (arbitrary) index value of zero. Thus we generated sets of ordinal categorical data; since the data were generated from continuous distributions the bin indices of each attribute were ordered. Further, the number of possible categories for each datum is given by N_B.

3.2 Experiments

We have repeated the following experiments for $D = 2, 4, 6$ and for $N_B = 6, 9, 12$ making a total of nine experiments on the synthetic ordinal data. Since the trends are clear and identical across all experiments, for the sake of brevity, we show only results for $D = 2, N_B = 6$; $D = 4, N_B = 9$; $D = 6, N_B = 6$ and $D = 6, N_B = 12$. In each experiment we have shuffled different numbers of the attributes and observed the minimum test error obtainable for both ways of handling the categorical attributes (integer mapping and indicator variables). Shuffling removes the ordering in the data and by shuffling more attributes, more of the ordering can be removed. In this way we are able to vary the characteristics of the dataset from completely ordered (i.e. ordinal) in the case of no shuffling, to completely nominal after every attribute has been shuffled. The results shown are all averages over ten random initialisations of the population; the sizes of the error bars over the ten repetitions imply that this is a sufficient number of trials to obtain reliable results. In addition, for the shuffling experiments, each shuffled attribute was re-ordered in a different way on each repetition.

The results are shown in Figures 1–4 where we also show error bars so that it is possible to gauge the significance of the variations. The filled circle is the error obtained for the continuous data whereas the open circles are the errors for the integer mapping strategy. It is clear that even when the categories are unshuffled, that discretisation has increased the error since this process removes information from the data. Shuffling the categorical attributes clearly increases the error (open circles); in addition, the variance of the errors increases since some of the random shuffles preserve more order than others leading to a large variability in the observed error rates.

Using indicator variables, on the other hand (filled triangles), the errors for completely unshuffled data are worse than for integer mapping. But the error rate from using indicator variables is insensitive to shuffling of the attributes; the error variances for indicator variables are also little affected by shuffling. When around half of the attributes have been shuffled, indicator variables give consistently lower errors than integers. Across all nine sets of experiments, it

thus seems clear that for ordinal data, integer mapping gives the lowest errors but for categorical data, indicator variables give lower errors.

Figure 3 shows a particularly interesting extension to these data. The open triangles show the errors from representing each shuffled attribute with an indicator variables and the remaining unshuffled attributes with integer mappings. This hybrid representation yields better errors than using all integers or all indicators, further reinforcing the observations above. (All nine experiments show the same unambiguous trend – we only show these 'hybrid' results in Figure 3 because in all other graphs showing the additional data creates so much clutter that the main results are obscured.)

A numerical summary of the results for all nine experiments is given in Table 1.

Table 1. Summary of misclassification errors for shuffling the synthetic ordinal data

		Unshuffled						Shuffled			
D	N_B	Continuous		Integers		Indicator		Integers		Indicator	
		Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
2	6	0.1215	0.001	0.1338	0.003	0.1743	0.013	0.2745	0.070	0.1708	0.010
4	6	0.1527	0.001	0.1580	0.001	0.1999	0.011	0.2833	0.039	0.2093	0.009
6	6	0.1689	0.002	0.1900	0.006	0.2388	0.006	0.3089	0.038	0.2400	0.018
2	9	0.1215	0.001	0.1370	0.001	0.1735	0.009	0.2913	0.066	0.1948	0.035
4	9	0.1527	0.001	0.1579	0.001	0.2188	0.012	0.3140	0.040	0.2206	0.018
6	9	0.1689	0.002	0.1917	0.010	0.2653	0.017	0.3300	0.040	0.2766	0.018
2	12	0.1215	0.001	0.1318	0.002	0.2503	0.051	0.3133	0.056	0.2415	0.050
4	12	0.1527	0.001	0.1589	0.001	0.2276	0.019	0.3243	0.048	0.2291	0.011
6	12	0.1689	0.002	0.1865	0.013	0.2664	0.013	0.3467	0.043	0.2655	0.014

4 Synthetic Nominal Dataset

The results in the preceding section show that when the order in ordinal data is progressively destroyed by shuffling, indicator variables become an increasingly better choice of attribute representation. In this section, we seek to generate synthetic *nominal* data and compare integer mapping and indicator variables. This represents a complementary approach to that of the previous section.

4.1 Dataset

Again we have generated data with either $D = 2, 4, 6$ attributes and the numbers of categories in each attribute is given by $N_B = 6, 9, 12$. We have randomly generated the different categories according to the percentage schema: For $N_B = 6$: Class 1 $= \{5, 5, 10, 20, 25, 35\}$ and Class 2 $= \{35, 25, 20, 10, 5, 5\}$. For $N_B = 9$: Class 1 $= \{3, 3, 3, 11, 10, 10, 20, 20, 20\}$ and Class 2$=\{20, 20, 20, 10, 10, 11, 3, 3, 3\}$. And for $N_B = 12$: Class 1 $= \{2, 2, 2, 2, 8, 8, 8, 8, 15, 15, 15, 15\}$ and Class 2 $= \{15, 15, 15, 15, 8, 8, 8, 8, 2, 2, 2, 2\}$. That is, for $N_B = 6$ and Class 1, for example, 5% of the data were drawn from category 1, 5% from category 2, 10% from

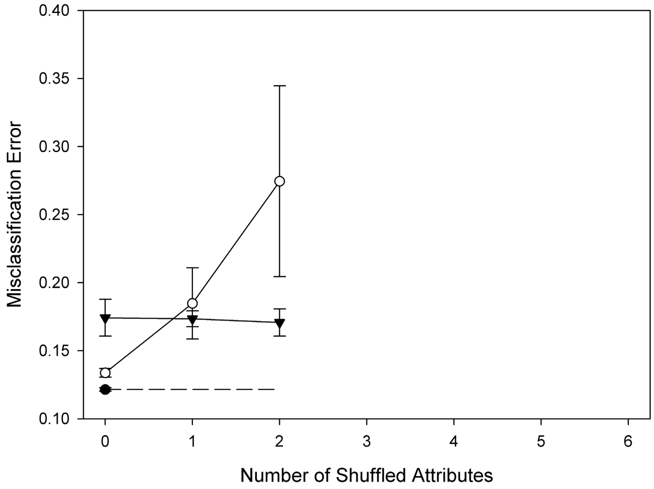


Fig. 1. Misclassification error vs. Number of Shuffled Attributes; $D = 2, N_B = 6$. The dashed line shows the mean baseline error for continuous variables. Filled circle/dashed line = baseline error for continuous attributes. Open circles = integer mapping. Filled triangles = indicator variables. See text for details.

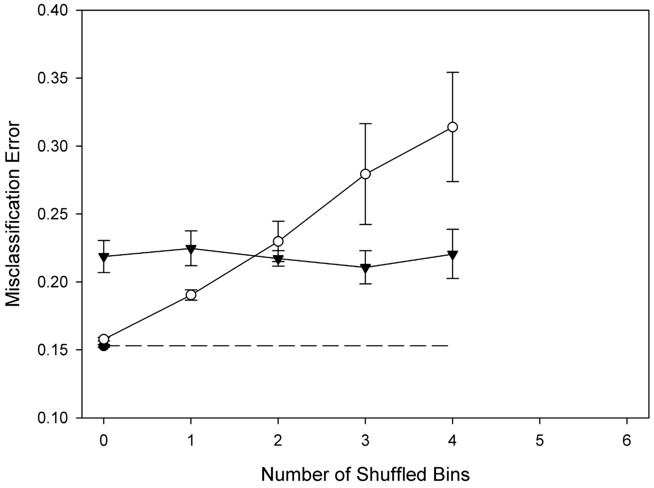


Fig. 2. Misclassification error vs. Number of Shuffled Attributes; $D = 4, N_B = 9$. The dashed line shows the mean baseline error for continuous variables. Filled circle/dashed line = baseline error for continuous attributes. Open circles = integer mapping. Filled triangles = indicator variables. See text for details.

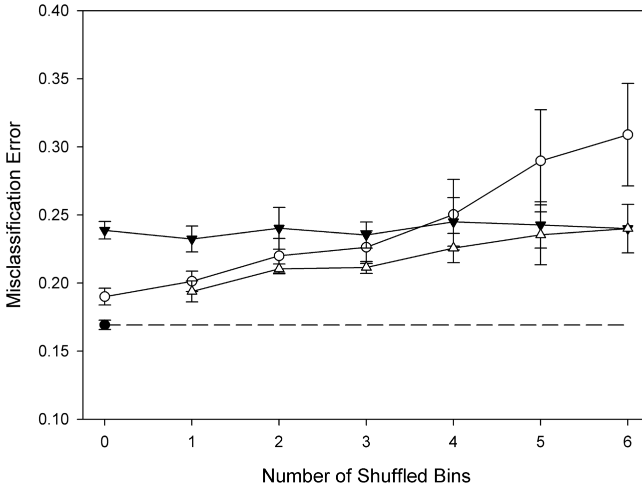


Fig. 3. Misclassification error vs. Number of Shuffled Attributes; $D = 6, N_B = 6$. The dashed line shows the mean baseline error for continuous variables. Filled circle/dashed line = baseline error for continuous attributes. Open circles = integer mapping. Filled triangles = indicator variables. See text for details.

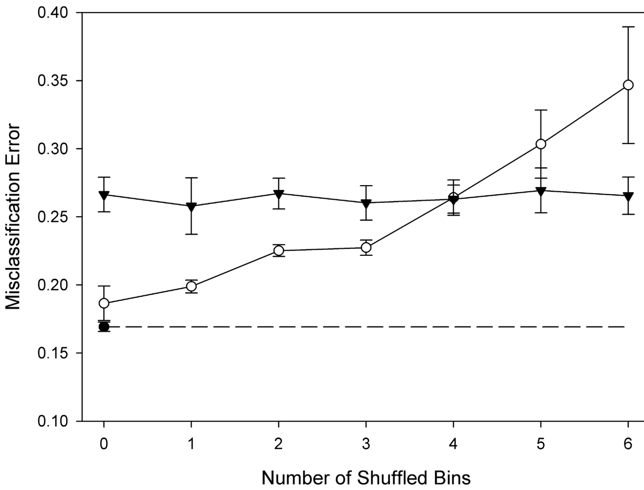


Fig. 4. Misclassification error vs. Number of Shuffled Attributes; $D = 6, N_B = 12$. The dashed line shows the mean baseline error for continuous variables. Filled circle/dashed line = baseline error for continuous attributes. Open circles = integer mapping. Filled triangles = indicator variables. See text for details.

category 3, 20% from category 3, 25% from category 4 and 35% from category 5. These nominal data were shuffled further for each of the ten tests. As with the synthetic ordinal data, we generated $10 \times D$ training data per class and $100 \times D$ test data per class.

4.2 Experiments

The results for ten repetitions of classifying each dataset using either integer mappings of the (nominal) attributes or indicator values are shown in Table 2. From Table 2 – and wholly consistent with the results in Section 3 – we can conclude that for all nine datasets, indicator variables produce lower error rates than integer mapping for these purely nominal attributes.

Table 2. Summary of misclassification errors on the synthetic nominal data. The lowest errors are shown in bold face.

		Integers		Indicators	
D	N_B	Mean	SD	Mean	SD
6	6	0.1803	0.0314	**0.1084**	0.0108
4	6	0.1936	0.0225	**0.1463**	0.0148
2	6	0.2288	0.0361	**0.1910**	0.0065
6	9	0.2174	0.0237	**0.1339**	0.0086
4	9	0.2455	0.0394	**0.1675**	0.0083
2	9	0.2503	0.0334	**0.1858**	0.0102
6	12	0.2798	0.0435	**0.1539**	0.0149
4	12	0.2908	0.0540	**0.1710**	0.0127
2	12	0.2855	0.0461	**0.2010**	0.0126

5 Real Datasets

Finally we consider a range of real datasets containing either solely categorical attributes or mixed continuous/categorical attributes. These datasets, which are publicly-available from the UCI Repository[1], are summarised in Table 3; some have previously been investigated in [10]. For those datasets which have a separate *test* set we have measured error performance using that test set. For the rest, we have have employed Alpaydin's F-test [1,2] to gauge the statistical significance of any differences in misclassification error. The results for the train-test datasets are shown in Table 4. Indicator variables perform best on the three Monk datasets which is consistent with the fact that these data are nominal. Interestingly, the test error on the Monk-3 dataset is *less* than the training error: this comes about because in the Monk-3 training set 5% of the patterns have been deliberately mislabelled by the originators of this dataset. Since the evolved classifier gives zero error on the test set, we infer we are obtaining good generalisation.

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

For the Adult dataset, however, integers perform best . Since this is a mixed dataset, we cannot be sure of the relevance to classification of the categorical attributes; further it is difficult to decide whether some of the attributes are ordered or not. The Highest Level of Education Achieved attribute ∈*Bachelors, Some- college, 11th, High-School-grad, etc.*, for example, clearly has a recognised *hierarchy* but whether or not this is an order is not straightforward.

Table 3. Summary of the real datasets used

	Classes	Attributes			Dataset Sizes		
		Total	Categorical	Continuous	Whole	Train	Test
Mushroom	2	22	22	0	5644	50%	50%
Tic Tac Toe	2	9	9	0	958	50%	50%
Australian Credit	2	14	6	8	690	50%	50%
Monk-1	2	6	6	0	556	124	432
Monk-2	2	6	6	0	601	169	432
Monk-3	2	6	6	0	554	122	432
Adult	2	14	6	8	45222	30162	15060

Table 4. Summary of results for the train-and-test real datasets. Lowest errors are shown in bold face.

		Integers				Indicator Variables			
	#Evaluations	Train		Validation		Train		Validation	
Monk-1	50000	0.000	0.000	**0.000**	0.000	0.000	0.000	**0.000**	0.000
Monk-2	50000	0.178	0.031	0.251	0.027	0.068	0.046	**0.102**	0.065
Monk-3	50000	0.050	0.009	0.027	0.000	0.035	0.009	**0.000**	0.000
Adult	20000	0.162	0.005	**0.163**	0.005	0.174	0.008	0.173	0.008

Results for the datasets compared using the 5×2 *cv* test [1] are shown in Table 5; an F value > 4.74 is statistically significant at the 95% level. Again for the purely nominal categorical Mushroom and Tic-Tac-Toe datasets, indicator variables give statistically better performance. The mixed Australian Credit dataset gives a better error with integers. Unfortunately, in the case of this last dataset all information on the attributes has been removed (we believe as a condition of allowing publication of the dataset). Consequently, we can say little more about the superiority of integers in this case.

Some of these real datasets were also investigated by Loveard & Ciesielski [10] who considered two other possible approaches for mapping categorical variables. Direct comparison needs some care since Loveard & Ciesielski presented the average of the best five runs they obtained whereas we have shown the average over all of our 10 runs; further details of how the datasets were split for cross-validation are missing from [10]. Nonetheless, we both obtain zero error for the Mushroom dataset. For the Adult dataset, Loveard & Ciesielski obtained a best average of 0.149 using indicator variables ("binary conversion" in their

parlance) whereas we observe a best average error of 0.163 for integer mapping ("numeric conversion" to Loveard & Ciesielski). This discrepancy may be due to Loveard & Ciesielski presenting only their best hand-picked results compared to our uncensored reporting.

For the Australian Credit data, Loveard & Ciesielski obtain a best average error of 0.139 for integer mapping whereas we obtain an error of 0.119, also for integer mapping. (We cannot, of course, judge the statistical significance between these results because there are insufficient details in [10].)

Table 5. Summary of results for the 5×2 cv tests on real datasets. Lowest validation errors and the statistically significant values of F are shown in bold face. An F statistic > 4.74 is statistically significant at the 95% level.

		Integers		Indicator Variables		
	Evaluations	Train	Validation	Train	Validation	F
Mushroom	30000	0.012	0.012	0.000	**0.000**	4.233
Tic-Tac-Toe	50000	0.195	0.232	0.137	**0.175**	**4.757**
Australian credit	20000	0.090	**0.119**	0.092	0.131	**5.563**

6 Conclusions

In this paper we have compared the misclassification performance of classifiers evolved by multiobjective genetic programming for two different methods of handling categorical attributes: mapping-to-integer and indicator variables. The mapping-to-integer approach implies the attributes exist in a metric space which is reasonable for ordinal attributes but unwarranted for purely nominal attributes.

We have constructed two sets of synthetic data: a nominal dataset and an ordinal dataset. For the nominal data, indicator variables perform best whereas for ordinal data, integer mapping is superior. By shuffling the ordinal data by increasing degrees, we have progressively removed ordering in this data – effectively 'nominal-ising' it – and observed that when more than around half the attributes had been shuffled, indicator variables yielded the best error rates.

The results from purely categorical real data show that indicator variables are better although the results from the mixed continuous/categorical data appear confounded by uncertainties about the discriminatory rôle of the continuous variables and the interpretation of the categorical attributes.

Based on the results from the synthetic datasets, we conclude that ordinal attributes should be represented by integer mapping and nominal attributes by indicator variables. In practice, however, determining to which type a categorical attribute belongs is sometimes not straightforward: perhaps our conclusion could also be used for data exploration in that the 'type' of a categorical attribute could be *inferred* from the representation which gives the best error.

References

1. Alpaydin, E.: Combined 5×2 cv F-test for comparing supervised classification learning algorithms. Neural Computation 11, 1885–1892 (1999)
2. Dietterich, T.: Approximate statistical tests for comparing supervised classification learning algorithms. Neural Computation 10, 1895–1923 (1998)
3. Ekárt, A., Németh, S.Z.: Selection based on the Pareto nondomination criterion for controlling code growth in genetic programming. Genetic Programming & Evolvable Machines 2, 61–73 (2001)
4. Fonseca, C.M., Fleming, P.J.: Multi-objective optimization and multiple constraints handling with evolutionary algorithms. Part 1: A unified formulation. IEEE Trans. Systems, Man & Cybernetics 28, 26–37 (1998)
5. Guo, H., Jack, L.B., Nandi, A.K.: Feature generation using genetic programming with application to fault classification. IEEE Transactions on Systems, Man & Cybernetics - Part B 35, 89–99 (2005)
6. Ito, T., Iba, H., Sato, S.: Non-destructive depth-dependent crossover for genetic programming. In: 1st European Workshop on Genetic Programming, Paris, France, pp. 14–15 (1998)
7. Krawiec, K.: Genetic programming-based construction of features for machine learning and knowledge discovery tasks. Genetic Programming & Evolvable Machines 3, 329–343 (2002)
8. Kumar, R., Rockett, P.: Improved sampling of the Pareto-front in multi-objective genetic optimization by steady-state evolution: A Pareto converging genetic algorithm. Evolutionary Computation 10, 283–314 (2002)
9. Loveard, T., Ciesielski, V.: Representing classification problems in genetic programming. In: Congress on Evolutionary Computation, Seoul, Korea, pp. 1070–1077 (2001)
10. Loveard, T., Ciesielski, V.: Employing nominal attributes in classification using genetic programming. In: 4th Asia-Pacific Conference on Simulated Evolution and Learning (SEAL 2002), Singapore, pp. 487–491 (2002)
11. Smith, M.G., Bull, L.: Genetic programming with a genetic algorithm for feature construction and selection. Genetic Programming & Evolvable Machines 6, 265–281 (2005)
12. Tian, Y., Deng, N.: Support vector classification with nominal attributes. In: Hao, Y., Liu, J., Wang, Y.-P., Cheung, Y.-m., Yin, H., Jiao, L., Ma, J., Jiao, Y.-C. (eds.) CIS 2005. LNCS (LNAI), vol. 3801, pp. 586–591. Springer, Heidelberg (2005)
13. Zhang, Y., Rockett, P.I.: Evolving optimal feature extraction using multi-objective genetic programming: A methodology and preliminary study on edge detection. In: Genetic & Evolutionary Computation Conference (GECCO 2005), Washington, DC, pp. 795–802 (2005)

The Effects of Constant Neutrality on Performance and Problem Hardness in GP

Edgar Galván-López, Stephen Dignum, and Riccardo Poli

Department of Computing and Electronic Systems, University of Essex, Colchester, UK
{egalva,sandig,rpoli}@essex.ac.uk

Abstract. The neutral theory of molecular evolution and the associated notion of neutrality have interested many researchers in Evolutionary Computation. The hope is that the presence of neutrality can aid evolution. However, despite the vast number of publications on neutrality, there is still a big controversy on its effects. The aim of this paper is to clarify under what circumstances neutrality could aid Genetic Programming using the traditional representation (i.e. tree-like structures) . For this purpose, we use fitness distance correlation as a measure of hardness. In addition we have conducted extensive empirical experimentation to corroborate the fitness distance correlation predictions. This has been done using two test problems with very different landscape features that represent two extreme cases where the different effects of neutrality can be emphasised. Finally, we study the distances between individuals and global optimum to understand how neutrality affects evolution (at least with the one proposed in this paper).

1 Introduction

Evolutionary Computation (EC) systems are inspired by the theory of natural evolution. The theory argues that through the process of selection, organisms become adapted to their environments and this is the result of accumulative beneficial mutations. However, in the late 1960s, Kimura [11] put forward the theory that the majority of evolutionary changes at molecular level are the result of random fixation of selectively neutral mutations. Kimura's theory, called the *neutral theory of molecular evolution* or, more frequently, the *neutral theory*, considers a mutation from one gene to another as neutral if this modification does not affect the phenotype.

Neutral theory has inspired researchers from the EC community to incorporate neutrality in their systems in the hope that it can aid evolution. Despite the vast work carried out towards understanding the effects of neutrality in evolutionary search, as will be seen in the following section, there are no general conclusions on its effects.

In this paper we make an effort to understand how neutrality works and identify under what circumstances it could aid Genetic Programming (GP) [13].

The paper is organised as follows. In the next section, previous work on neutrality in EC is summarised. In Section 3, the approach used to carry out our research is described. In Section 4 we review the notion of fitness distance correlation (*fdc*) and present the distance used to calculated *fdc* in tree-like structures. In Section 5 we introduce the problems used to analyse the effects of neutrality. In Sections 6 and 7 we present and discuss the results of experiments with unimodal and deceptive landscape problems and draw some conclusions.

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 312–324, 2008.
© Springer-Verlag Berlin Heidelberg 2008

2 Previous Work on Neutrality

As mentioned previously, there has been a considerable amount of work on neutrality. However, there is a lack of final conclusions on its effects. For instance, Fonseca and Correia [7] developed two redundant representations using different approaches based on mathematical tools. The authors focused their attention on the properties highlighted by Rothlauf and Goldberg [17] and pointed out some potential fallacies in such work. In [17], Rothlauf and Goldberg stated that when using synonymously redundant representations, the landscape connectivity is not increased. Fonseca and Correia, however, stated that this is not necessarily true. They reported that in their proposed representations the connectivity tends to increase accordingly to the number of redundant bits.

In [5], an effort has been made to analyse some aspects of the search space in GP. Ebner focused his attention on the fact that finding a given behaviour on such search spaces is not as difficult as finding a given individual. As the author pointed out, this is due to in GP using tree-like structures, many individuals map exactly to the same phenotype. Correspondingly, the same situation is observed in nature (i.e., highly redundancy). With these elements in hand, Ebner suggested that search spaces that present similar characteristics as the ones found in nature may be beneficial in evolving potential solutions towards finding a global solution to a specific problem.

This line of thought was further explored by Ebner *et al.* [6]. The authors pointed out that the presence of neutrality could aid evolution under certain circumstances. To illustrate this point,they used two types of mappings: random Boolean networks (RBNs) and cellular automaton (see [6] for a full description of both mappings). In their studies, Ebner *et al.* pointed out that neutrality could have a positive impact in evolutionary search, if a given population is spread out in a neutral network. To exemplify this point, Ebner *et al.* used a dynamic fitness landscape and shown how the presence of neutrality gives the opportunity to a population to ”start” over again and eventually (if this is the case) being able to escape from local optima (which is not the case on a non-redundant mapping). In [12], Knowles and Watson distinguished advantages and disantages of what they called random genetic redundancy. In particular the authors used RBNs [6] to conduct their experiments. In their work, Knowles and Watson mentioned that in particular RBNs are useful because help to mantain diversity. On the other hand, the authors also mentioned that the performance, in some cases, was better when neutrality introduced by RBNs was not present and so, one should be carefull when adding artificial neutrality.

Yu and Miller [22] argued that in the traditional GP representation, implicit neutrality is difficult to identify and control during evolution and so, they used Cartesian GP (CGP) to add what they called *explicit* neutrality. To analyse the effects of neutrality they tested their approach on the even parity problems and used an Evolutionary Strategy. CGP uses a genotype-phenotype mapping that allows programs to have inactive code (i.e., this is how neutrality is artificially added) at genotype level. The genotype uses an integer string coding. This type of encoding allowed the authors to use Hamming distance to measure the amount of neutrality present in the evolutionary search. In their studies, they found that the larger the amount of neutrality present during evolution, the higher the percentage of success in finding the global optimum, regardless the mutation rate. They concluded that neutrality is fundamental to improve evolvability.

Collins [4] claimed that Yu and Miller's conclusions in [22] were flawed. Collins started his analysis by highlighting that the use of a Boolean parity problem is a strange choice given that the problem in itself is neutral (i.e., the fitness value of individuals is the same except for the one that finds the global optimum) and, so, the effects of neutrality are harder to analyse using this type of problem. Moreover, Collins focused his attention on the results found for the even-12-parity Boolean problem and pointed out that the CGP representation used in [22] favours shorter sequences than those yielding solutions for this problem. He also showed that the good results reported in [22] (i.e., 55% of success in finding the solutions) are not surprising and that random search has a better performance. He also concluded that the effects of neutrality are more complex than previously thought.

Theoretical work has also been developed in an effort to shed some light on neutrality. In [8] we studied perhaps the simplest possible form of neutrality using GAs: a neutral network of constant fitness, identically distributed in the whole search space. For this form of neutrality, we analysed both problem-solving performance and population flows. We used the fitness distance correlation, calculating it in such a way to make the dependency between problem difficulty and neutrality of the encoding explicit.

In [2], Beaudoin *et al.* proposed a family of fitness landscapes called the *ND* landscapes, where one can vary the length of the genome *N* and the neutral degree distribution *D*. This presents some advantages over other types of landscapes (i.e., *NKp*, *NKq* and *Technological*) previously proposed in the literature to analyse neutrality, which, however, do not consider the distribution of neutrality. This, according to the authors, is instead a key feature in evolution.

Recently, we [14] proposed and studied three different types of genotype-phenotype encodings that add neutrality in the evolutionary search. To analyse in detail the effects of these kinds of neutrality on three different types of landscape, we used the *fdc* and the newly introduced notion of *phenotypic mutation rates*. We also developed a mathematical framework that helped explain some of our empirical findings.

As it can be seen from the previous summaries, the results reported on the effects of neutrality in evolutionary search are very mixed (except perhaps the theoretical works previously summarised).

3 Constant Neutrality

With many primitive sets, GP has the ability to create a rich and complex network of natural networks. This may be a useful feature, but it is a feature that is hard to control and analyse. For this reason, in this paper we propose to artificially create a further neutral network within the search space, which is simple and entirely under our control, thereby making it possible to evaluate the effects of neutrality on GP behaviour and performance. In particular, we propose to add a neutral network of constant fitness. More specifically:

– In our approach, called *constant neutrality*, neutrality is "plugged" into the traditional GP representation (i.e., tree-like structures) by adding a flag to the representation: when the flag is set, the individual is on the neutral network and, as indicated

previously, its fitness has a pre-fixed value. When the flag is off, the fitness of the individual is determined as usual.

- We use fitness distance correlation (*fdc*) as a measure of hardness when neutrality is present in the evolutionary search and in its absence. We also perform extensive empirical experiments to corroborate the results found by *fdc*.
- We use two benchmark problems with significantly different landscape features: a unimodal landscape where we expect neutrality to be detrimental and a multimodal deceptive landscape where neutrality helps evolution by escaping from local optima.

As mentioned previously, to allow the presence of constant neutrality in the GP process, we added a flag to the representation. This is in charge of indicating if a given individual is or is not on the neutral layer. To allow the migration to and from the neutral layer we use a special mutation, which is applied with probability P_{nm}. The process to set or unset the neutral flag works as follows. Firstly, we initialise the population by creating random individuals in the usual way, and, with probability P_{nm}, we activate the flag of the resulting individuals. Secondly, during the evolutionary process, every time an offspring is created, it inherits the flag of its parent. However, before the individual is inserted in the population, with probability P_{nm}, we flip its flag. For example, if the parent of the individual was already on the neutral layer (i.e., flag activated) and the flag is flipped, the offspring is off the neutral layer and its fitness is calculated as usual. If, instead, the flag was not flipped, the individual's flag remains activated and its fitness is constant. The situation is symmetric if the original parent was not on the neutral layer.

In the proposed approach, we used traditional crossover (i.e., swapping subtrees) and structural mutation [20] and so, to add neutrality using any of these operators, we set the value of $P_{nm} > 0$ (see Section 5). The main reason of using structural mutation in our experiments is mainly because it is more close to the defition of distances between tree-based structures widely discussed in [19,20]. Structural mutation involves two types of mutation: inflate and deflate mutation. Inflate mutation takes a random internal node whose arity a is lower than the maximum arity defined in the function set, and it replaces it with a random function of arity $a+1$. A terminal is inserted as the $(a+1)$ argument of the new function. Deflate mutation takes any internal node with an arity a greater than the minimum arity defined in the function set and where at least one argument is a terminal, and it replaces the node with a function of arity $a-1$, deleting one of the terminals rooted on the original node.

In the form of neutrality explained previously, we can easily see how the size of the search space has increased dramatically. However, we still are in the presence of a single global optimum. So, the addition of neutrality comes at a cost: after all we are expanding the search space without correspondingly expanding the solution space. Thus, we should expect that the presence of neutrality will aid evolution only if it modifies the bias of the search algorithm in such a way to make the sampling of the global optimum much more likely.

With these elements in hand, it is very difficult to imagine how adding neutrality to a unimodal landscape can aid evolution. The addition of the form of neutrality explained previously – *constant neutrality* – changes the unimodal landscape into a landscape with plateaus where the search becomes totally random, which could lead to not finding the

optimum solution. If the global solution is found, we should expect that it will take longer for the evolutionary process to find it because of the presence of plateaus.

At this point some question arise. What will the effects of neutrality be on multi-modal landscapes? Specifically, will the problem be easier in the presence of neutrality? Will neutrality provide a path to cross optima solutions and be able to find global solutions? Will the presence of neutrality provide advantages on tree-like structures, like for example, control bloat? To answer these questions, we will use fitness distance correlation (fdc) as a measure of hardness. Moreover we will conduct extensive empirical experiments to compare the performance of our approach and the findings of fdc.

4 Fitness Distance Correlation

Jones [10] proposed *fitness distance correlation* (fdc) to measure the difficulty of a problem by studying the relationship between fitness and distance. The idea behind *fdc* was to consider fitness functions as heuristics functions and to interpret their results as indicators of the distance to the nearest global optimum in the search space.

The definition of *fdc* is quite simple: given a set $F = \{f_1, f_2, ..., f_n\}$ of fitness values of n individuals and the corresponding set $D = \{d_1, d_2, ..., d_n\}$ of distances to the nearest global optimum, we compute the correlation coefficient r, as:

$$r = \frac{C_{FD}}{\sigma_F \sigma_D},$$

where:

$$C_{FD} = \frac{1}{n} \sum_{i=1}^{n} (f_i - \overline{f})(d_i - \overline{d})$$

is the covariance of F and D, and σ_F, σ_D, $\overline{f}$ and $\overline{d}$ are the standard deviations and means of F and D, respectively. The n individuals used to compute *fdc* can be chosen in different ways. For reasonably small search spaces or in theoretical calculations it is often possible to sample the whole search space. However, in most other cases, fdc is estimated by constructing the sets F and D via some form of random sampling.

According to [10] a problem can be classified in one of three classes: (1) *misleading* ($r \geq 0.15$), in which fitness tends to increase with the distance from the global optimum, (2) *difficult* ($-0.15 < r < 0.15$), for which there is no correlation between fitness and distance, and (3) *easy* ($r \leq -0.15$), in which fitness increases as the global optimum approaches. There are some known weaknesses with *fdc* as a measure of problem hardness [1,16]. However, it is fair to say that the method has been generally very successful [3,10,14,18,19,20,21].

Several papers have proposed the use of *fdc*. Slavov and Nikolaev were among the first to use *fdc* in GP [18]. In their experiments, they calculated *fdc* using fitness-distance pairs which were recorded during runs. The authors defined the distance between a given individual (DT) in the form of tree-like structure and the global optimum (O) as follows

$$d(DT,O) = \begin{cases} 1 + \sum_{i \in DT,O} d(child(DT_i), child(O_i)) & \text{if root } DT_i \neq \text{root } O_i, \\ \sum_{i \in DT,O} d(child(DT_i), child(O_i)) & \text{otherwise.} \end{cases}$$

Later, Clergue *et al.* [3] extended this idea. As a first step in their investigation, they used the same function and terminal sets defined by Punch *et al.* [15] where the function set was $F_{set} = \{A, B, C, \cdots\}$, where A has arity 1, B has arity 2 and so on. The terminal set included a single symbol, X. Moreover, they set a restriction and generated trees respecting one rule: for every node in a tree, if the arity of the node is n, the nodes below it must have an arity less than n. Initially, Clergue *et al.* defined the distance between trees T_1 and T_2 as follows $d_1(T_1, T_2) = |weight(T_1) - weight(T_2)|$ where $weight(T) = 1 \cdot n_X(T) + 2 \cdot n_A(T) + 3 \cdot n_B(T) + 4 \cdot n_C(T) + \cdots$, $n_X(T)$ is the number of symbols X in the tree T, $n_A(T)$ is the number of symbols A in the tree T and so on. However, this definition of distance between a pair of trees had a major problem: two trees with very different structures can have distance of 0. In an effort to overcome this problem, Clergue *et al.* came up with following idea. Each tree with root i must have a greater weight than the trees with root j, if $j < i$, where: (a) $i, j \in \{X, A, B, C, \cdots\}$ and (b) there is an order such that $X < A < B < C \cdots$. Moreover, a prize is given to each root. This new definition improved the situation but there was still a problem: two individuals that have vertical axis of symmetry have a distance 0, despite their structures being different.

The same authors eventually overcame these limitations [3,21] and computed and defined a distance (which is the distance used in this work) between two trees in three stages: (a) The trees are overlapped at the root node and this process is recursively applied starting from the leftmost subtrees, (b) For each pair of nodes at matching positions, the difference of their codes c (i.e., index of an instruction within the primitive set) is calculated and (c) The computed differences are combined in a weighted sum. That is, the distance between two trees T_1 and T_2 with roots R_1 and R_2 is calculated as follows:

$$dist(T_1, T_2, k) = d(R_1, R_2) + k \sum_{i=1}^{m} dist(child_i(R_1), child_i(R_2), \frac{k}{2}) \tag{1}$$

where $d(R_1, R_2) = (|c(R_1) - c(R_2)|)^z$. $child_i(Y)$ is the ith of the m possible subtrees of a generic node Y, if $i \leq m$, or the empty tree otherwise, and c evaluated on the root of an empty tree is equal to 0. Finally, k is a constant used to give different weights to nodes belonging to different levels in the trees. This distance produced successful results on a wide variety of problems [19,20,21].

Calculating distances between trees is not as simple as it is when the distance is calculated for a pair of bitstrings. Once the distance has been computed between two trees, it is necessary to normalise it in the range $[0, 1]$. In [20], Vanneschi proposed five different methods to normalised the distance. Here, we propose another way to carry out more efficiently this task and that we have called "normalisation by maximum distance using a fair sampling". This works as follows: (a) A sample of n_s individuals is created using the ramped half and half method and using a global maximum depth greater than the maximum depth allowed during evolution, (b) The distance is calculated between each individual belonging to n_s and the global optimum, (c) Once all the distances have been calculated, the maximum distance m_s found in the sampling is stored.

At the end of this process, the global maximum distance m_s is used to normalise the distances throughout the evolutionary process. The global maximum depth[1] used to

[1] For our experiments the maximum distance is given by $maximum_depth + 2$.

create a sample of individuals n_s[2] is greater than the maximum depth allowed through evolution. So, through the evolutionary process a higher value for the global maximum distance it is highly unlikely to be found. Moreover, to control bloat we allow a maximum length that is determined during the application of the sampling method.

In the following section, we calculate fdc using Equation (1) on the problems used to study the problem hardness in the absence and in the presence of neutrality in evolutionary search.

5 Experimental Setup

We have used two problems to analyse neutrality. The first one is the Max problem. The problem consists of finding a program, subject to size or depth (D), which produces the largest possible output. For this problem we have defined $F = \{+\}$, $T = \{0.5\}$ and maximum depth $D = 5$ (for all our examples the root node is at depth 0). Naturally, using these sets, the problem has only one global optimum (a full tree with depth 5) and the landscape is unimodal. For this example, we have used the grow method [13] to create our population.

The second problem is a trap function [9]. For this example, we have used the function:

$$f(X) = \begin{cases} 1 - \frac{d}{B} & \text{if } d \leq B, \\ \frac{R(d-B)}{1-B} & \text{otherwise} \end{cases}$$

where d is the normalised distance between a given individual and the global optimum solution. d, B and R are values in the range of $[0,1]$. B is the slope-change location for the optima and R sets their relative importance. For our problem, there is only one global optimum and by varying the parameters B and R, we make the problem easier or harder.

Figure 1 depicts the global optimum solution used in the trap problem where $B = 0.01$ and $R = 0.8$ (i.e., the problem is considered to be very difficult). The language that has been used to code individuals in the trap function is the one proposed by Punch *et al.* [15]. Their idea was to use functions with one increasing arity. That is, for a function set $F = \{A, B, C, \cdots\}$ their corresponding arities are 1, 2, 3, $\cdots$ and the terminal set is defined by $T = \{x\}$ which its arity is 0. Moreover, we have initialised our individuals with the full method [13] using $D = 5$. The maximum allowed depth for programs was 7. Notice that we have used two different methods to initialise the populations for each of the two examples. This has been done to avoid sampling the global solutions for both problems.

The experiments were conducted using a GP with tournament selection size 10. We used standard crossover and mutation (inflate and deflate) independently (i.e., when crossover was used, mutation did not take place during evolution and *vice versa*). To obtain statistically meaningful results, we performed 100 independent runs for each of the values of fitness of the neutral layer. Runs were stopped when the maximum number of generations was reached. The parameters we have used for both problems are summarised in Table 1. In Tables 2 and 3 we show the constant value (f_n) assigned to the neutral layer for each of the problems.

[2] For our experiments n_s is typically 10 times larger than the population size.

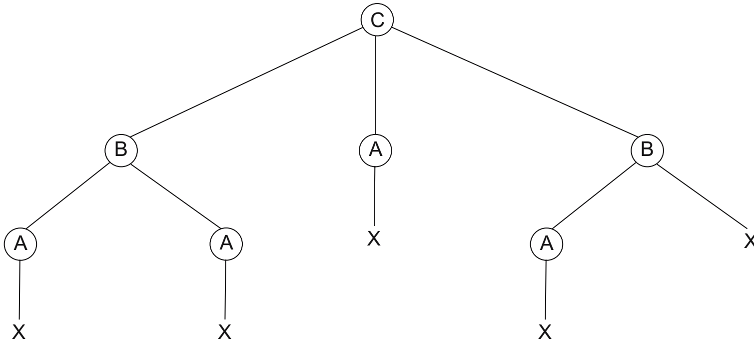


Fig. 1. A tree used as global optimum in the trap function setting $B = 0.01$ and $R = 0.8$ meaning that the problem is considered to be very difficult

Table 1. Summary of Parameters

Parameter	*Value*
Population Size	400
Generations	300
Neutral Mutation Probability (P_{nm})	0.05
Mutation Rate	90%
Crossover Rate	90%

6 Results and Analysis

6.1 Performance Comparison

Let us focus our attention on the Max problem (see Table 2). As discussed previously, it is very hard to imagine how neutrality could aid evolution in a unimodal landscape. When neutrality is not present, GP is able to find the global solution without difficulties both when using crossover and when using structural mutation, the percentage of success being 100% regardless the operator used. This situation, however, changes radically when neutrality is added in the evolutionary search. That is, when neutrality is added the performance of GP decreases. As shown in Table 2, the percentage of success goes from 100% when neutrality is not present to 0% when the fitness of constant neutrality is set to 15 (remember that for this problem the global optimum has fitness 16). This is easy to explain because as discussed previously, individuals which fitness is below the fitness of the neutral layer will tend to move there and once they are in the neutral layer, the search will behave like random search. Note that fdc correctly predicts these performance variations.

Now, let us consider the second problem – the trap function. In Table 3 we show the results found on this problem when calculating fdc. Again we have complemented this by comparing the performance of GP in the presence and in the absence of neutrality using standard crossover and structural mutation.

Table 2. Statistical information on the Max problem using $F = \{+\}$, $T = \{0.5\}$ and $D = 5$. The fitness of the global optimum is 16. Avr. Gen. refers to the average number of generations required to find the global optimum.

f_n value	fdc	*Crossover*		*Structural Mutation*	
		Avr. Gen	% Suc.	Avr. Gen	% Suc.
No neutrality	-0.9999	29.14	100%	14.22	100%
5	-0.1994	65.69	95%	17.08	100%
10	0.0661	350.29	17%	28.94	100%
15	0.1380	NA	0%	42.08	100%

Table 3. Statistical information on the Trap function using as global optimum the program shown in Figure 1

f_n value	fdc	*Crossover*		*Structural Mutation*	
		Avr. Gen	% Suc.	Avr. Gen	% Suc.
No neutrality	0.9971	5.00	1%	3.60	5%
0.10	0.9627	6.00	1%	3.25	3%
0.20	0.8638	6.00	1%	5.33	3%
0.30	0.7070	64.41	12%	107.75	4%
0.40	0.5677	66.83	12%	4.00	2%
0.50	0.4616	94.87	8%	NA	0%
0.60	0.3828	202.20	5%	NA	0%
0.70	0.3234	202.80	5%	NA	0%
0.80	0.2778	470.00	1%	NA	0%
0.90	0.2419	NA	0%	NA	0%

When neutrality is not present in the evolutionary search, we can see how fdc classify the problem as very difficult (i.e., $fdc = 0.9999$), which is actually the case. When neutrality is added, there are some circumstances where its presence is more helpful than others. For instance, when the constant fitness in the neutral layer is 0.30 and 0.40, the performance of GP increases dramatically when using standard crossover (i.e., when neutrality is not present, the percentage of success is only 1% compared with 12% when neutrality is added). By how much neutrality will help the search strongly depends on the constant fitness assigned in the neutral layer, f_n. However, for almost all values of f_n we observe improvements over the case where neutrality is absent when crossover is used. Here there is a rough agreement between fdc and actual performance.

Surprisingly, however, when structural mutation is used, in virtually all cases the addition of neutrality hinders performance, and there appears to be a general trend indicating that the higher f_n the worse the results. This goes exactly in the opposite direction of the predictions of fdc. This is perhaps the result of the distance in Equation (1) not being well-suited to capture the offspring-parent differences produced by the actions of the mutation operator.

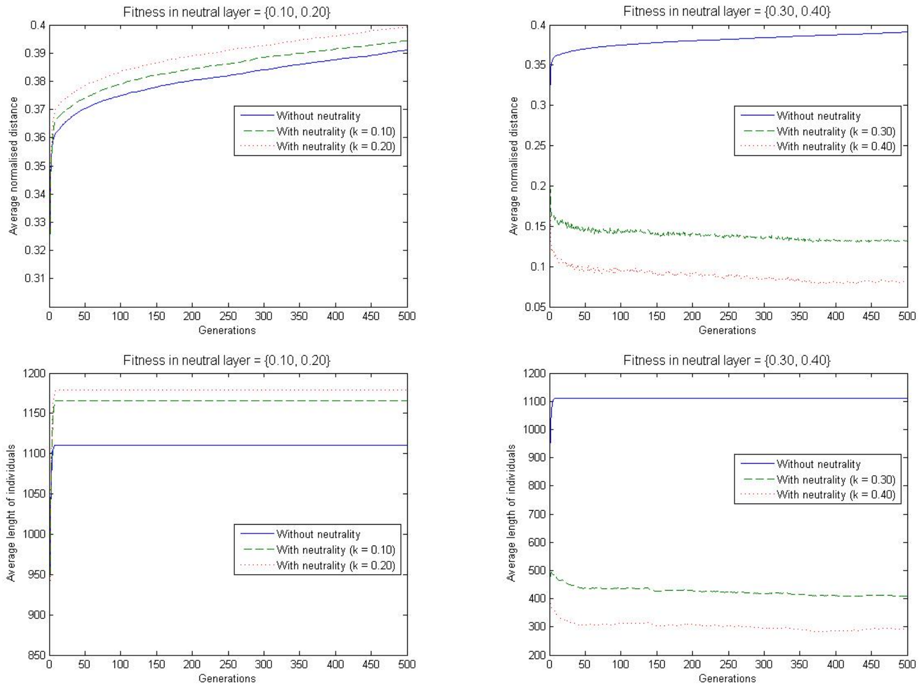


Fig. 2. Average normalised distance (top) and average length of individuals (bottom) using crossover on a difficult trap function. Figure 1 shows the global optimum.

6.2 Distances between Individuals and Global Optimum

As shown previously, neutrality aids evolution in deceptive landscapes when using crossover. This situation, however, varies depending on the constant value assigned to the neutral layer. Since GP with crossover is effectively the standard, we want to analyse in more detail how neutrality affects evolution. To do so, we study how the distance between the individuals in the population and the global optimum (shown in Figure 1) varies generation after generation for different values of f_n.

In the top left-hand side of Figure 2, notice how the normalised distance between individuals and the global optimum for the cases $f_n = \{0.10, 0.20\}$ effectively varies in the same ways as when neutrality is not present in evolution. Indeed, as confirmed by results shown in Table 3 the percentage of success are almost the same (i.e., in the range of 0% and 1%).

This situation, however, changes radically when using fitter neutral layers, i.e., $f_n = \{0.30, 0.40\}$, as shown at the top right-hand side of Figure 2. Notice how the average distance between individuals and the global optimum tends to drop dramatically compared to when $f_n \leq 0.20$. Individuals are now on average much closer to the global optimum and, so, it is easier for the GP system to eventually sample it and solve the problem.

A reason why GP with crossover is able to sample the global optimum more often in the presence of neutrality with $f_n = \{0.30, 0.40\}$ is that GP tends to produce shorter encodings. This can be observed in the bottom right-hand side of Figure 2. However, this does not mean that GP produces in general smaller individuals regardless the constant fitness set in the neutral layer.

7 Conclusions

The effects of neutrality are unclear. The goal of this paper is to clarify under what circumstances neutrality could aid GP evolution.

In this paper we considered perhaps the simplest possible form of neutrality in GP. This is introduced simply by adding a flag to each individual which indicates whether or not the individual is on the neutral layer. We used the distance, shown in Equation (1), proposed and studied in [19,20,21] to calculate fitness distance correlations (fdc) in GP landscapes. We used it as a measure of hardness and compared its findings with extensive empirical experimentation using the Max problem with unimodal landscape features and a Trap function with deceptive landscape features.

We found that fdc roughly predicts how problem difficulty is affected by the presence of neutrality for GP with subtree crossover. The prediction of difficulty for GP with structural mutation is, instead, more problematic.

Based on these observations and on empirical results, it is clear that the form of neutrality studied in this paper (constant neutrality) can only aid evolution when the landscape is complex and multimodal. This is interesting, since, in fact, most realistic GP landscapes present such features. However, we have also found that it is important how to set the fitness of the neutral layer (f_n) carefully if for the potential benefits of neutrality to materialise. Much less we can say about the problems where GP with structural mutation could benefit from the use of constant neutrality.

References

1. Altenberg, L.: Fitness Distance Correlation Analysis: An Instructive Counterexample. In: Proceedings of the Seventh International Conference on Genetic Algorithms, pp. 57–64. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1997)
2. Beaudoin, W., Verel, S., Escazut, C.: Deceptivennes and Neutrality The ND Family of Fitness Landscapes. In: Keijzer, M., Cattolico, M., Arnold, D., Babovic, V., Blum, C., Bosman, P., Butz, M.V., Coello Coello, C., Dasgupta, D., Ficici, S.G., Foster, J., Hernandez-Aguirre, A., Hornby, G., Lipson, H., McMinn, P., Moore, J., Raidl, G., Rothlauf, F., Ryan, C., Thierens, D. (eds.) Proceedings of the 2006 Conference on Genetic and Evolutionary Computation, Seattle, WA, USA, july 8-12, 2006, pp. 505–514. ACM Press, New York (2006)
3. Clergue, M., Collard, P., Tomassini, M., Vanneschi, L.: Fitness Distance Correlation and Problem Difficulty for Genetic Programming. In: Langdon, W.B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M.A., Schultz, A.C., Miller, J.F., Burke, E., Jonoska, N. (eds.) GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, july 9-13, 2002, pp. 724–732. Morgan Kaufmann Publishers, San Francisco (2002)

4. Collins, M.: Finding Needles in Haystacks is Harder with Neutrality. In: Beyer, H.-G., O'Reilly, U.-M., Arnold, D.V., Banzhaf, W., Blum, C., Bonabeau, E.W., Cantu-Paz, E., Dasgupta, D., Deb, K., Foster, J.A., de Jong, E.D., Lipson, H., Llora, X., Mancoridis, S., Pelikan, M., Raidl, G.R., Soule, T., Tyrrell, A.M., Watson, J.-P., Zitzler, E. (eds.) GECCO 2005: Proceedings of the 2005 Conference on Genetic and evolutionary computation, Washington DC, USA, june 25-29, 2005, vol. 2, pp. 1613–1618. ACM Press, New York (2005)
5. Ebner, M.: On the Search Space of Genetic Programming and its Relation to Nature's Search Space. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 1999), Washington, D.C., USA, July 6-9, 1999, vol. 2, pp. 1357–1361. IEEE Computer Society Press, Los Alamitos (1999)
6. Ebner, M., Shackleton, M., Shipman, R.: How Neutral Networks Influence Evolvability. Complexity 7(2), 19–33 (2001)
7. Fonseca, C., Correia, M.: Developing Redudant Binary Representations for Genetic Search. In: Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC 2005), Edinburgh, september 2-4, 2005, pp. 372–379. IEEE Computer Society Press, Los Alamitos (2005)
8. Galván-López, E., Poli, R.: Some Steps Towards Understanding How Neutrality Affects Evolutionary Search. In: Runarsson, T.P., Beyer, H.-G., Burke, E., Merelo-Guervós, J.J., Whitley, L.D., Yao, X. (eds.) PPSN 2006. LNCS, vol. 4193, pp. 778–787. Springer, Heidelberg (2006)
9. Goldberg, D.E., Deb, K., Horn, J.: Massive Multimodality, Deception, and Genetic Algorithms. In: Männer, R., Manderick, B. (eds.) Parallel Problem Solving from Nature, vol. 2, pp. 37–48. Elsevier Science Publishers, BV, Amsterdam (1992)
10. Jones, T.: Evolutionary Algorithms, Fitness Landscapes and Search. PhD thesis, University of New Mexico, Albuquerque (1995)
11. Kimura, M.: Evolutionary Rate at the Molecular Level. Nature 217, 624–626 (1968)
12. Knowles, J.D., Watson, R.A.: On the Utility of Redundant Encodings in Mutation-Based Evolutionary Search. In: Guervós, J.J.M., Adamidis, P.A., Beyer, H.-G., Fernández-Villacañas, J.-L., Schwefel, H.-P. (eds.) PPSN 2002. LNCS, vol. 2439, pp. 88–98. Springer, Heidelberg (2002)
13. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge (1992)
14. Poli, R., Galván-López, E.: On the Effects of Bit-Wise Neutrality on Fitness Distance Correlation, Phenotypic Mutation Rates and Problem Hardness. In: Stephens, C.R., Toussaint, M., Whitley, L.D., Stadler, P.F. (eds.) FOGA 2007. LNCS, vol. 4436, Springer, Heidelberg (2007)
15. Punch, B., Zongker, D., Goodman, E.: The Royal Tree Problem, A Benchmark for Single and Multiple Population Genetic Programming. In: Angeline, P., Kinnear, K. (eds.) Advances in Genetic Prorgamming, 1996, vol. 2, pp. 299–316. MIT Press, Cambridge, MA (1996)
16. Quick, R.J., Rayward-Smith, V.J., Smith, G.D.: Fitness Distance Correlation and Ridge Functions. In: Eiben, A.E., Bäck, T., Schoenauer, M., Schwefel, H.-P. (eds.) PPSN 1998. LNCS, vol. 1498, pp. 77–86. Springer, Heidelberg (1998)
17. Rothlauf, F., Goldberg, D.: Redundant Representations in Evolutionary Algorithms. Evolutionary Computation 11(4), 381–415 (2003)
18. Slavov, V., Nikolaev, N.I.: Fitness Landscapes and Inductive Genetic Programming. In: Smith, G.D., Steele, N.C., Albrecht, R.F. (eds.) Artificial Neural Nets and Genetic Algorithms: Proceedings of the International Conference, ICANNGA 1997, University of East Anglia, Norwich, UK, Springer, Heidelberg (1997)
19. Tomassini, M., Vanneschi, L., Collard, P., Clergue, M.: A Study of Fitness Distance Correlation as a Difficulty Measure in Genetic Programming. Evolutionary Computation 13(2), 213–239 (2005)

20. Vanneschi, L.: Theory and Practice for Efficient Genetic Programming. PhD thesis, University of Lausanne, Switzerland (2004)
21. Vanneschi, L., Tomassini, M., Clergue, M., Collard, P.: Difficulty of Unimodal and Multimodal Landscapes in Genetic Programming. In: Cantú-Paz, E., Foster, J.A., Deb, K., Davis, L., Roy, R., O'Reilly, U.-M., Beyer, H.-G., Kendall, G., Wilson, S.W., Harman, M., Wegener, J., Dasgupta, D., Potter, M.A., Schultz, A., Dowsland, K.A., Jonoska, N., Miller, J., Standish, R.K. (eds.) GECCO 2003. LNCS, vol. 2724, pp. 1788–1799. Springer, Heidelberg (2003)
22. Yu, T., Miller, J.F.: Needles in haystacks are not hard to find with neutrality. In: Foster, J.A., Lutton, E., Miller, J., Ryan, C., Tettamanzi, A.G.B. (eds.) EuroGP 2002. LNCS, vol. 2278, pp. 13–25. Springer, Heidelberg (2002)

Applying Cost-Sensitive Multiobjective Genetic Programming to Feature Extraction for Spam E-mail Filtering

Yang Zhang[1], HongYu Li[2], Mahesan Niranjan[2], and Peter Rockett[1]

[1] Laboratory for Information and Vision Engineering, Department of Electronic and Electrical Engineering, The University of Sheffield, Mappin Street, Sheffield S1 3JD, UK
hegallis@gmail.com, p.rockett@shef.ac.uk

[2] Department of Computer Science, The University of Sheffield, Regent Court, 211 Portobello Street, Sheffield S1 4DP, UK
hongyu.cn@gmail.com, m.niranjan@dcs.shef.ac.uk

Abstract. In this paper we apply multiobjective genetic programming to the cost-sensitive classification task of labelling spam e-mails. We consider three publicly-available spam corpora and make comparison with both support vector machines and naïve Bayes classifiers, both of which are held to perform well on the spam filtering problem. We find that for the high cost ratios of practical interest, our cost-sensitive multiobjective genetic programming gives the best results across a range of performance measures.

1 Introduction

Spam – unsolicited e-mail – is a major and growing problem. It is estimated that in the month of May 2006, for example, 86% of all e-mails sent were spam [17]. Consequently, a large amount of effort has been expended on devising effective filters to identify spam e-mails.

Spam filtering presents a number of challenges. To incorrectly label as spam any solicited e-mails is highly undesirable; on the other hand, it is preferable to filter as much spam as possible. Conventional rule-based spam filters are usually implemented by maintaining/updating a list of keywords or phrases as indicators of unsolicited e-mails. Rejection or acceptance of a given e-mail is decided based on matching to this list. Unfortunately, the dynamic content of spam and individual tolerances to different content are difficult to interpret as reusable and updateable rules. Hence the rule-based method has been criticised for lacking good "time characteristics", since is hard to manually adapt the rules to changing spam; furthermore, there is no automatic learning mechanism which can be utilised. In contrast, content-based (or statistical-based) spam filtering has attracted increasing interest and much work has been carried-out on the design of spam filters using data mining and machine learning methods [5,9,19,20,21].

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 325–336, 2008.
© Springer-Verlag Berlin Heidelberg 2008

Among the methods employed, Bayesian analysis [19] is widely used to implement statistically-based spam filters. Based on the assumptions that the distributions of both non-spam and spam e-mails are known beforehand and components in the representation vector of an e-mail are independent, a naïve Bayesian (NB) classifier becomes an attractive approach due to its simplicity. However, as pointed-out by Fawcett [7], both the distributions, priors and concept are changing over different time spans. The first assumption can hardly be correct in real world circumstances. Further, the independence assumption of the feature vector components can only be met by applying further feature extraction methods, e.g. Independent Component Analysis. In addition, aware of the Bayesian word analysis, spammers started to attach or mix their messages with common words to break the underpinning independence assumptions, so distorting the real information in spam e-mails.

The dependence on the assumption of class distributions is undesirable given the skewed and drifting class distributions for both classes, particularly for spam e-mails, together with the fact that spammers are intelligent humans who deliberately produce 'cleverer' spam to avoid being recognised by the available filters. Longer term, a continuous learning mechanism is desired to track the changing spam e-mail concept.

Support vector machines (SVMs) have been a popular method for designing spam filters via a machine learning approach [5,13,22]. Li & Niranjan [16] have also explored subspace representations of this, and similar high dimensional problems, using sequential forward selection and backward deletion. Other conventional classifiers such as decision trees and neural networks have been used in the spam filtering problem as well as genetic programming (GP). GP was introduced to evolve a text classifier by Clack et al. [4] who fed documents (including e-mails) to a central classifier which autonomously routed documents to different categories. A single fitness function was designed based on weighted and combined terms. Hirsch et al. [10] used GP to evolve compact rules for document classification giving equal weight to precision and recall as a single fitness value. Katirai [12] applied GP to filter spam e-mails by evolving a specific classifier to distinguish between spam and legitimate e-mails. A single fitness value was calculated based on the raw combination of precision and recall over the training dataset. Katirai concluded that although the precision of the GP output was comparable to that of the naïve Bayes classifier, its recall trailed naïve Bayes by 6.67%.

From a technical point of view, spam filtering – the devising classifiers to label an e-mail as spam or non-spam – presents a number of challenges. In this, as in many pattern recognition problems (e.g. medical diagnosis), the costs of erroneous labelling are not equal so simply minimising the raw error rate is not satisfactory: here the unequal costs have to be taken into account. Also, the input feature vectors are typically of very high dimensionality. This is a significant obstacle to machine learning tasks involving small numbers of training patterns per dimension due to the limited representation of the problem by the available samples – high-dimensional spaces are inherently sparse. Without care, the normal learning process will tend to produce a system with poor generalisation.

In this paper we report the results of applying multiobjective GP to the cost-sensitive problem of spam filtering (csMOGP). We have examined three publicly available spam corpora and made quantitative comparison with support vector machines and the naïve Bayes classifier models which are held to perform well on the spam filtering task. For the practically important cases of high cost ratios, csMOGP is shown to out-perform both comparator algorithms.

2 Spam Datasets

There are only a few publicly-available benchmark datasets for spam filtering evaluation. Though publishing a collection of spam e-mails is not difficult, privacy reasons prevent the publication of non-spam e-mails. Hence many studies report results from private corpora collected for specific purposes which makes fair comparisons between different algorithms difficult. Furthermore, choosing the appropriate performance measures with which to conduct the comparisons is another important research issue [3,21], especially when cost-sensitive learning is concerned (see Section 3.4). For these reasons, three publicly available datasets were chosen for this study: LingSpam, Spambase and SpamAssassin.

LingSpam[1]. LingSpam is a popular spam e-mail filtering corpus consisting of 481 spam messages and 2,412 messages sent via the Linguist list (a list about the profession and science of linguistics). In this corpus the legitimate messages are more topic-specific which is reflected by the relatively lower spam ratio than the other two datasets used here.

Spambase[2]. Spambase, constructed in 1999, is a collection of 4,601 vectors with each of them representing either a spam or legitimate message using 57 pre-selected attributes, most of them words or character frequencies. The original messages in the Spambase corpus are not available.

SpamAssassin[3]. The SpamAssassin mail corpus was publicly released in 2005. It comprises groups of private e-mails donated by different users. Unlike the previous two corpora, all the e-mail headers are reproduced in full in SpamAssassin. There are 6,047 messages in the corpus, with 1,897 spam e-mails (a 31.4% spam ratio), 3,900 easy non-spam e-mails which do not contain any of the obvious 'spam' signatures (such as HTML tags, etc.) and 250 hard but legitimate e-mails which are 'closer' to typical spam in many respects (use of HTML, unusual HTML mark-up, coloured text, 'spammish'-sounding phrases, etc.)

In this study words are taken as being separated by white space. In the Spambase corpus, individual characters such as '$' and '#' are used; in the other two corpora, those characters are not considered. Further, e-mails may contain web links, HTML scripts, pictures, attachments, etc. In this study, the focus is on the text content of messages.

Details of the three corpora used in this work are summarised in Table 1.

[1] From http://www.aueb.gr/users/ion/data/lingspam/public.tar.gz
[2] From http://www.ics.uci.edu/~mlearn/MLRepository.html
[3] From http://spamassassin.apache.org

Table 1. Summary of the spam e-mail corpora used giving sizes and spam ratios: LingSpam, Spambase and SpamAssassin

Corpus	Spam	Non-spam	Total	Spam ratio	Vocabulary	Published	Encrypted
LingSpam	481	2,412	2,893	16.63%	65,728	2000	No
Spambase	1,813	2,788	4,601	39.4%	–	1999	Yes
Spam Assassin	1,897	4,150	6,047	31.4%	134,850	2005	No

3 Feature Extraction/Classifier Construction Using GP

The *feature selection* stages used in common inductive learning algorithms appear incapable of providing sufficient discriminability for classification or can be easily bypassed by spammers. After detailed investigation of the dynamic properties of spam, Fawcett [7] concluded that more effort should be spent in devising effective *feature extraction* methods which should be able to adapt to the new distorted instances via automatic data-driven learning. Zhang et al. [22] confirmed that certain feature selection/extraction methods can enhance classification performance. They argued that especially when cost ratios are considered, the choice of the appropriate feature pre-processing step can be critical.

More generally, it is widely accepted that mapping a vector of raw pattern attributes into a new domain can improve the separability of classes. (Indeed, many algorithms which are usually recognised as *classifiers* fall into in the paradigm of projecting the raw pattern attributes into $\mathbb{R}^1$ and then assigning a class label by thresholding in the projected space. Fisher's linear discriminant is probably the most obvious example.)

In this paper we have used multiobjective genetic programming (MOGP) to induce the 'optimal' mappings from $\mathbb{R}^m \rightarrow \mathbb{R}^1$, where m is the dimension of the pattern vector for the spam labelling problem. We view this as *feature extraction* followed by a (trivial) thresholding step in $\mathbb{R}^1$ to assign a class label since the principal focus of our work has been inducing the mapping. (This could equally well be viewed as directly inducing a classifier.)

3.1 Raw Pattern Vector Construction

The spam filtering problem has its origin in text categorisation so the content-based machine learning approach regards the words in an e-mail as the original features. Thus a vector can be constructed for each e-mail. If we consider the content of the original e-mail, a vector representation can be constructed by analysing the various words in the e-mails. An intuitive way of constructing the vector for each e-mail is to use the number of times a word appears – Term Frequency (TF) – as the components of the vector. However, a word which has a high frequency of occurrence is not always a good signature of the e-mail's content if it occurs in many other e-mails. Thus, enhancements to the TF representation, Document Frequency (DF) and Inverse DF (IDF) are used and multiplied by the TF, and termed $TF - IDF$. DF is the number of times

a word occurs in *all* e-mails in the corpus and is a popular weighting method in text categorisation. IDF is defined as:

$$IDF(e_i) = \log\left(\frac{\#D}{DF(e_i)}\right)$$

where $DF(e_i)$ is the document frequency of word, i in the collection. $\#D$ is the number of e-mails in the corpus. The pattern vectors, $\mathbf{x}_j$ were normalised such that:

$$x_{i,j} = \frac{TF(e_{i,j}) \times IDF(e_j)}{\sqrt{\sum_{j=1}^{n} TF^2(e_{i,j}) \times IDF^2(e_j)}}$$

Yang & Pedersen [21] concluded that DF has clear advantages over both mutual information (MI) and term strength (TF) representations while Drucker et al. [5] have demonstrated the effectiveness of $TF-IDF$ representations in spam filtering. Thus this method is used to construct raw pattern vectors which form the input to both the csMOGP feature extraction stage and the conventional comparator classifiers.

We have performed a pre-processing sequence on the corpora which we believe is fairly standard in text categorisation and spam filtering. For each e-mail, its textual portion was represented by a concatenation of the subject line and the body of the message. The word-extraction process was carried-out by substituting all non-alphanumeric characters with white spaces; HTML tags were removed. Words are defined as contiguous strings of characters delimited by white space with all characters converted to lowercase for simplicity. We preprocessed the e-mail messages by stop word elimination and Porter stemming [18]. Further, Zipf's law was applied to eliminate word features with frequency less than five in either class. Finally, the vectorisation process was completed by weighting the words using $TF - IDF$.

The dimensionalities of the final pre-processed pattern vectors were: 505 for LingSpam, 57 for Spambase and 3644 for SpamAssassin.

3.2 Description of MOGP System

The genetic programming implementation used in this work is the steady-state evolutionary strategy [14] straightforwardly adapted for GP. Depth-fair crossover [11] was used along with mutation; since the evolutionary strategy was steady-state, crossover followed by mutation was always applied. The MOGP settings are listed in Table 2. Optimisation was carried-out for a fixed number of function evaluations after which the evolution was stopped. As listed in Table 2, three objectives were used to construct a three-dimensional fitness vector, comprising: tree complexity, misclassification cost and an approximation to the Bayes error in the 1D projected space.

The tree complexity objective is designed to inhibit bloat for which it has been found to be very effective [6]. We have used a simple count of tree nodes

as a complexity measure. In addition, a beneficial side-effect of minimising the tree complexity is to minimise the numbers of terminal nodes used in the tree, that is the numbers of raw pattern attributes used. Thus, as well as evolving the 'optimal' feature extraction mapping, we also perform implicit *feature selection*; unless an attribute has sufficient discriminatory power it will tend not be included (in a tree of a given size).

The misclassification cost objective was designed to enable cost-sensitive learning and since it is pivotal to this work, it is discussed in greater detail in the following sub-section.

The Bayes error estimate in the transformed feature space was calculated by histogramming the projected class-conditioned values and calculating the overlap between the spam and non-spam classes. Bayes error is a fundamental lower bound on the separability of classes and was used to provide a selective pressure to 'force apart' the two class-conditioned distributions in the decision space. (We have also observed empirically that Bayes error acts as a useful 'helper' objective in the early stages of evolution and thus can speed convergence.)

Having projected the patterns into a 1D decision space, we determined the optimum decision threshold for minimum misclassification cost using Golden Section search.

All three objectives were optimised simultaneously within a Pareto framework using the method of Fonseca & Fleming [8], leading to an (approximation to the) Pareto set for a given dataset.

Table 2. Cost-Sensitive MOGP Settings

Terminal set	Input pattern vector elements 10 floating point numbers $\in [0.0, 0.1, \ldots 0.9, 1.0]$
Function set	sqrt, log, pow2, unary minus, sin, -, +, ×, ÷, max, min, if-then-else
Raw fitness vector	Bayes error, Misclassification cost, Number of tree nodes
Population size	500
Initial population	50% full trees + 50% random trees
Original tree depth	7
Max. number of generations	500,000
Stopping criterion	Max no. of generations exceeded

3.3 Cost-Sensitive Multiobjective Feature Extraction

The spam filter design problem is naturally multiobjective. Firstly, given a spam filter, stopping as many as the spam e-mails as possible is in direct conflict with preventing the filtering of legitimate e-mails. In fact, the conflicting nature of decreasing the number of false positives (fraction labelled as spam from the non-spam class) and the increasing the number of true positives (fraction labelled correctly as spam) is a very general problem in pattern recognition. In other

words, it is difficult to design a filter which simultaneously optimises the *precision* and *recall*.

Secondly, as studies on the cost-sensitive spam filtering problem have shown – see [3] – the trade-off between increasing precision and recall becomes more complex when different cost ratios are considered. Indeed, the use of GP in cost-sensitive classification has been previously addressed by Li et al. [15] who defined a single aggregate fitness function calculated from precision and recall with the cost ratio.

Let the cost of mislabelling a spam e-mail as a legitimate class be $C_{S\rightarrow\ L}$ and the cost of mislabelling a legitimate e-mail as spam be $C_{L\rightarrow\ S}$, with the correct label incurring zero cost. $N_{S\rightarrow\ L}$ and $N_{L\rightarrow\ S}$ are the numbers of e-mails which are labelled as legitimate-from-spam and labelled as spam-from-legitimate class, respectively. The misclassification cost, MC is calculated as:

$$MC = C_{L\rightarrow\ S} \times \frac{N_{L\rightarrow\ S}}{N_L} \times P_L + C_{S\rightarrow\ L} \times \frac{N_{S\rightarrow\ L}}{N_S} \times P_S$$

where P_S and P_L are the priors, estimated from the training set. N_S and N_L are the total numbers of spam and legitimate e-mails over the training set.

3.4 Performance Measures

To make a fair comparison is always critical in performance evaluation and a number of performance measures have been proposed [3,21]; this is harder when different cost ratios have to be taken into account for a typical cost-sensitive problem such as spam filtering. In this study, the precision, P and recall, R pair is calculated for each algorithm under every investigated cost ratio and for each corpus. (Precision and recall comparisons have been carried out in [12,21].) Furthermore, for each combination, the raw misclassification error rate and weighted accuracy ($WAcc$) [8] are calculated to compare not only the raw error rate but also the algorithm's performance when false negatives and false positives are weighted differently under different cost ratios. The same evaluation process has been employed in [9,22] and in [3] where detailed descriptions of the measures can be found.

$N_{S\rightarrow\ S}$ is the number of spam samples correctly labelled as spam and $N_{L\rightarrow\ L}$ the number of correctly labelled legitimate e-mails. Precision, P, recall, R and weighted accuracy, $WAcc$ are defined by:

$$P = \frac{N_{S\rightarrow\ S}}{N_{S\rightarrow\ S} + N_{L\rightarrow\ S}} \quad R = \frac{N_{S\rightarrow\ S}}{N_{S\rightarrow\ S} + N_{S\rightarrow\ L}} \quad WAcc_\mu = \frac{\mu N_{L\rightarrow\ L} + N_{S\rightarrow\ S}}{\mu N_L + N_S}$$

where $\mu = C_{L\rightarrow\ S}/C_{S\rightarrow\ L}$ denotes the cost ratio. Following the suggestions in [2], three different values have been examined here: $\mu = \{1, 9, 999\}$. With increasing μ, the penalty on false positives – mislabelling legitimate-to-spam – increases. To give a reasonable interpretation of the $WAcc$ value when the μ value is very high, $TCR_\mu = N_S/(\mu N_{L\rightarrow\ S} + N_{S\rightarrow\ L})$ was introduced [2] to act as a single measure, larger values indicating better spam filtering performance.

Zhang et al. [22] conducted their comparisons via pairwise combinations of the five classifiers under investigation using TCR_9 values. Hitherto, the usual practice to gauge the statistical difference between the performance of different classifiers has been to perform N-fold cross-validation followed by a t-test. This, however, is unsound due to the failure of the implicit independence assumptions. Consequently, we have used Alpaydin's F-test [1] to statistically compare classifier performance where the F-measure was calculated for TCR values with different μ values to decide whether or not to reject the null hypothesis that the performances of the two spam filters were identical. Throughout this work we used a 95% confidence level to infer a statistical difference. The comparisons are focused on SVM and NB since they are widely considered to be the best performing on spam filters [21,22]. The implementations used were from the Weka Machine Learning system[4] with parameters of each classifier 'tuned' to minimise misclassification cost. The cost-sensitive learning for both SVM and NB was implemented by weighting the training instances according to the cost ratio per class.

4 Results

To assess the relative performances of the classifiers, each corpus was split into equal-sized training and validation sets. The results reported here are the averages over ten folds for each corpus. Since MO optimisation yields a Pareto set of equivalent solutions, we have selected a single csMOGP-generated classifier for comparison from the minimum misclassification cost over the validation set.

In Table 3 the raw mislabelled patterns are counted directly without consideration of the class from which they come. Viewed as a pure pattern recognition problem ($\mu = 1$), csMOGP appears superior on the LingSpam corpus and competitive with SVM on Spambase although is out-performed by both SVM and NB on SpamAssassin. (Note, raw error rate alone does not to address the cost-sensitive learning problem here and these results are shown only for the sake of completeness; we have not analysed them further.)

We believe that in practice, both users and organisations would tend to err on the side of receiving a few more spam e-mails rather than risk losing legitimate e-mails which might be mislabelled as spam. Thus, in the following analyses we place greater emphasis on the results for larger values of μ – that is, imposing a large cost on a legitimate-to-spam misclassification.

The precision/recall pairs in Table 4 show that as μ increases (the penalty of misclassifying a legitimate e-mail as spam increases), then recall decreases. In other words, because of the increasing cost of losing a legitimate e-mail, the discrimination process can achieve the greatest profit by minimising the number of legitimate e-mails mislabelled as spam. The SVM gives infinite precision and no recall on LingSpam and Spambase for $\mu = 999$, which means that for a very unbalanced cost ratio, the SVM classified all patterns as legitimate e-mails which is not useful in practice since it is not filtering-out any spam. In contrast,

[4] See http://www.cs.waikato.ac.nz/ml/weka/

Table 3. Raw Error Rates Comparisons among SVM, Naïve Bayes and csMOGP

Classifier	SVM			Naïve Bayes			csMOGP		
Cost Ratio	1:1	1:9	1:999	1:1	1:9	1:999	1:1	1:9	1:999
LingSpam	3.5%	1.6%	16.6%	3.7%	3.1%	3.0%	2.9%	5.0%	5.1%
Spambase	11.0%	28.2%	39.4%	20.7%	20.5%	19.8%	10.4%	19.7%	19.7%
SpamAssassin	1.8%	27.4%	31.2%	3.7%	10.2%	10.2%	7.4%	10.1%	18.1%

csMOGP is able to give 100% precision for the high cost ratio of 1:999, which means no legitimate e-mail is wrongly labelled as spam.

Zhang et al. [22] have pointed-out that since the precision/recall values cannot provide comparisons which incorporate the const-sensitive penalty information, we have calculated the $WAcc$ (Table 5) and TCR (Table 6) measures to address this issue.

Table 4. Precision/Recall Comparisons among SVM, Naïve Bayes and csMOGP. Upper number is the precision, the lower recall. (∞ denotes infinite precision).

Classifier	SVM			Naïve Bayes			csMOGP		
Cost Ratio	1:1	1:9	1:999	1:1	1:9	1:999	1:1	1:9	1:999
LingSpam	1.00	0.92	∞	0.88	0.89	0.89	1.00	1.00	1.00
	0.80	0.99	0.0	0.90	0.93	0.93	0.90	0.83	0.60
Spambase	0.91	0.98	∞	0.67	0.67	0.68	0.88	0.98	1.00
	0.81	0.29	0.0	0.95	0.95	0.95	0.66	0.35	0.14
SpamAssassin	0.98	1.00	∞	0.99	0.76	0.76	0.89	0.98	1.00
	0.96	0.12	0.0	0.89	0.98	0.98	0.87	0.69	0.42

From Table 5 it is clear that csMOGP delivers most of the highest weighted accuracies across all combinations over the three corpora (13 highest over 18 pairwise comparisons, shown in bold). csMOGP gives the highest values for all corpora for $\mu = 999$. For $\mu = 9$, csMOGP gives the highest values of $WAcc$ for the SpamAssassin and Spambase corpora but for this cost ratio, SVM performs best on the LingSpam corpus although whether the difference (99.58% vs. 99.54%) is statistically significant is doubtful.

Table 5. Weighted Accuracy Comparisons for SVM, Naïve Bayes and csMOGP for Three Cost Ratios. Highest values shown in bold face.

Classifier	SVM			Naïve Bayes			csMOGP		
Cost Ratio	1:1	1:9	1:999	1:1	1:9	1:999	1:1	1:9	1:999
LingSpam	96.5%	**99.6%**	97.8%	96.3%	97.5%	97.7%	**97.1%**	99.5%	**99.9%**
Spambase	**89.1%**	94.8%	93.3%	79.3%	71.2%	70.7%	78.8%	**95.8%**	**99.9%**
SpamAssassin	**98.2%**	95.8%	99.9%	96.3%	86.7%	86.2%	91.6%	**97.9%**	**99.9%**

The set of values for the TCR measure are shown in Table 6. Here, higher values imply better spam filtering performance. csMOGP gives the highest TCR scores for all combinations with the exceptions of: i) $\mu = 9$ for the LingSpam corpus on which SVMs give a slightly greater value, and ii) $\mu = 1$ for the SpamAssassin corpus on which the SVM performs very well. (For reasons explained above, this latter case is not of great practical interest.) It is also noteworthy that $TCR < 1$ (the baseline value) denotes that the spam filter is unsuccessful in practice; clearly the NB classifier falls below this threshold for many values of cost ratio. To gauge the statistical significance of the differences between the classifiers' performance, we have conducted a series of F-tests, shown in Table 7 in which a circle denotes that csMOGP gives worse filtering performance over that corpus-cost combination and a tick denotes the superiority of csMOGP; a dash denotes no statistical difference at the 95% level. csMOGP offers statistically better performance than SVM and naïve Bayes over most of the corpus-cost combinations and universally so at the high cost ratios which are of practical importance.

Table 6. TCR Comparisons among SVM, Naïve Bayes and csMOGP for three cost ratios, $\mu = \{1, 9, 999\}$

Classifier	SVM			Naïve Bayes			csMOGP		
Cost Ratio	1:1	1:9	1:999	1:1	1:9	1:999	1:1	1:9	1:999
LingSpam	24.12	**25.66**	5.03	7.68	4.35	0.04	**35.2**1	23.63	**12.70**
Spambase	2.37	1.99	1.54	2.93	0.36	0.00	**4.31**	**2.44**	**1.66**
SpamAssassin	**37.82**	2.51	2.21	18.74	0.80	0.01	8.157	**5.14**	**3.80**

Table 7. Results of pairwise 5×2 *cv* F-tests between SVM, Naïve Bayes and csMOGP See text for details

Classifier	SVM			Naïve Bayes		
Cost Ratio	1:1	1:9	1:999	1:1	1:9	1:999
LingSpam	✓	–	✓	✓	✓	✓
Spambase	✓	✓	✓	✓	✓	✓
SpamAssassin	○	✓	✓	○	✓	✓

5 Discussion

From examination of typical trees it is apparent that the transformations generated by csMOGP are highly non-linear. With the increase in μ-value, the trees in the converged csMOGP evolutions are more complex. A number of intuitively 'sensitive' word stems have been selected by csMOGP, such as "adult", "monei", "edu", "free", "remov", "credit", etc. Specifically, the word "free" was been selected several times by csMOGP across a number of solutions.

Finally, the drifting concept and skewed class distributions in spam filtering present a challenge to conventional machine learning methods. These phenomena

stem from the fact that the people who produce spam are continually changing the topic as well as the way they deliver their information. Spammers are intelligent humans and they are continuously learning to cope with *de facto* spam filtering methods, even when machine learning techniques are applied. In other words, the spam problem together with the spammers are evolving, therefore machine learning approaches based on static learning will need retraining at various intervals during their lifetime. A few recent efforts have focused on dynamic learning as well as personalised spam filter design. csMOGP provides an obvious way to continuously 'co-evolve' with the spammers – the csMOGP-trained filter could be continuously updated (as a background process) using newly acquired training data. Examining the continuous evolution characteristics of csMOGP in spam filtering is a potentially promising area for future research.

6 Conclusions

We have examined the application of multiobjective genetic programming to the cost-sensitive task of spam filtering. In particular, we have evolved (near-)optimal feature extraction stages which map the content of the e-mail in question to a 1D decision space in which we apply a threshold to determine the class label. We have made comparison over three publicly-available spam corpora with support vector machines (SVMs) and naïve Bayes (NB) classifiers which are both held to be successful on the spam filtering task. For the practically important high cost ratios (which impose a high penalty of mislabelling a legitimate e-mail as spam), the csMOGP approach out-performs both SVMs and NB classifiers and for the case of the TCR metric, by statistically significant margins.

References

1. Alpaydin, E.: Combined 5×2 cv F-test for comparing supervised classification learning algorithms. Neural Computation 11, 1885–1892 (1999)
2. Androutsopoulos, I., Koutsias, J., Chandrinos, K., Paliouras, G., Spyropoulos, C.: An evaluation of naive Bayesian anti-spam filtering. In: Proc. Workshop on Machine Learning in the New Information Age, 11th European Conference on Machine Learning, pp. 9–17 (2000)
3. Androutsopoulos, I., Paliouras, G., Michelakis, E.: Learning to filter unsolicited commercial e-mail. NCSR Demokritos Technical Report No.2004/2 (2004)
4. Clack, C., Farrington, J., Lidwell, P., Yu, T.: Autonomous document classification for business. In: Proc. ACM Conf. AGENTS 1997, pp. 201–208 (1997)
5. Drucker, H., Wu, D., Vapnik, V.N.: Support vector machines for spam categorization. IEEE Trans. on Neural Networks 10, 1048–1054 (1999)
6. Ekárt, A., Németh, S.Z.: Selection based on the Pareto nondomination criterion for controlling code growth in genetic programming. Genetic Programming & Evolvable Machines 2, 61–73 (2001)
7. Fawcett, T.: In vivo spam filtering: A challenge problem for data mining. KDD Explorations 5, 140–148 (2003)

8. Fonseca, C.M., Fleming, P.J.: Multi-objective optimization and multiple constraints handling with evolutionary algorithms. Part 1: A unified formulation. IEEE Trans. Syst., Man & Cybern. 28, 26–37 (1998)
9. Hidalgo, J.G.: Evaluating cost-sensitive unsolicited bulk e-mail categorization. In: Proc. 17th ACM Symposium on Appl. Computing, pp. 615–620 (2002)
10. Hirsch, L., Saeedi, M., Hirsch, R.: Evolving rules for document classification. In: Keijzer, M., Tettamanzi, A.G.B., Collet, P., van Hemert, J.I., Tomassini, M. (eds.) EuroGP 2005. LNCS, vol. 3447, pp. 85–95. Springer, Heidelberg (2005)
11. Ito, T., Iba, H., Sato, S.: Non-destructive depth-dependent crossover for genetic programming. In: 1st European Workshop on Genetic Programming, pp. 14–15 (1998)
12. Katirai, H.: Filtering junk e-mail: A performance comparison between genetic programming and naïve Bayes (1999), available at, http://members.rogers.com/hoomank/katirai99filtering.pdf
13. Kolcz, A., Alspector, J.: SVM-based filtering of e-mail spam with content-specific misclassification costs. In: IEEE Int. Conf. on Data Mining,TextDM 2001 Workshop on Text Mining (2001)
14. Kumar, R., Rockett, P.: Improved sampling of the Pareto-front in multi-objective genetic optimization by steady-state evolution: A Pareto converging genetic algorithm. Evolutionary Computation 10, 283–314 (2002)
15. Li, J., Li, X., Yao, X.: Cost-sensitive classification with genetic programming. Congress on Evolutionary Computation 3, 2114–2121 (2005)
16. Li, H., Niranjan, M.: Discriminant subspaces of some high dimensional pattern recognition problems. In: IEEE Workshop on Machine Learning for Signal Processing, Thessaloniki (August 2007)
17. Lochart, A.: Quoted in Koprowski, G. J., Spam accounts for most e-mail traffic, Tech News World (2006), http://www.technewsworld.com/story/51055.html
18. Porter, M.: An algorithm for suffix stripping. Automated Library and Information Systems 4, 130–137 (1980)
19. Sahami, M., Dumais, S., Heckerman, D., Horvitz, E.: A Bayesian approach to filtering junk e-mail. In: AAAI Workshop on Learning for Text Categorization (1998)
20. Tretyakov, K.: Machine learning techniques in spam filtering. In: Data Mining Problem-oriented Seminar. MTAT.03., vol. 177, pp. 60–79 (2004)
21. Yang, Y., Pedersen, J.O.: A comparative study of feature selection in text categorization. In: Proc. 14th Int. Conf. on Machine Learning, pp. 412–420 (1997)
22. Zhang, L., Zhu, J., Yao, T.: An evaluation of statistical spam filtering techniques. ACM Transactions on Asian Language Information Processing (TALIP) 3, 243–269 (2004)

PlasmidPL: A Plasmid-Inspired Language for Genetic Programming

Lidia Yamamoto

Computer Science Department, University of Basel
Bernoullistrasse 16, CH-4056 Basel, Switzerland
lidia.yamamoto@unibas.ch

Abstract. We present PlasmidPL, a plasmid-inspired programming language designed for Genetic Programming (GP), and based on a chemical metaphor. The basic data structures in PlasmidPL are circular virtual molecules or rings which may contain code and data. Rings may react with each other to perform computations on the rings themselves. A virtual chemical reactor stochastically chooses which reactions should occur and when. Code and data may be rewritten in the process, leading to a system that constantly modifies itself. In order to be closer to chemistry, PlasmidPL relies solely on the data and code stored in molecules.

After describing the language, we show some hand-written sample programs that implement initial program generation, mutation and crossover within self-modifying chemical programs. These programs are then used to solve a typical symbolic regression problem, as a feasibility study. Finally, we discuss future directions into specific application scenarios that can benefit from such a chemical model.

1 Introduction

The motivation for this work lies in obtaining software that autonomously reacts to environmental changes by ultimately changing its own code. The applications for such self-evolving software include robotics [1,2], sensor-actuator networks [3], pervasive, organic and autonomic computing. As in biology, the software itself would be responsible for its own "survival", including reproduction and variation mechanisms which under selective pressure from the real world could result in successive generations of ever improving individuals.

PlasmidPL is a new artificial polymer chemistry [4] inspired by the structure and behaviour of plasmids. In biology, plasmids are small circular DNA segments that can exist and replicate separately from the chromosomal DNA of their host cell which is usually a bacterium. Multiple plasmids and several instances of the same plasmid may co-exist in the same cell.

A PlasmidPL program is a multiset of rings. Rings are circular arrays of atoms that can manipulate other rings, producing new rings as results. A ring data structure wraps around itself in a modulo fashion, such that any position in a ring is a valid position. In this way, information can always be extracted

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 337–349, 2008.
© Springer-Verlag Berlin Heidelberg 2008

from rings, or written to them without any "array out of bounds" exception. As with protected division in GP, the ring structure aims at helping to obtain valid programs using genetic operators. Furthermore, the language is designed with minimal syntactic and semantic constraints, such that any random program can potentially be interpreted to produce some result. In contrast with related approaches to the evolution of programs using chemical metaphors [5,6], computation with PlasmidPL relies exclusively on the data and code stored in molecules. Any kind of information storage outside molecules (such as external stacks, registers, or memory positions) is explicitly forbidden. The state of the system is therefore fully defined by the set of molecules present in the reactor.

Rings may be regarded either as passive data molecules or as standalone mini-threads of computation. They may react with each other to perform computations on the rings themselves. A virtual chemical reactor chooses which reactions should occur and when, acting as a thread scheduler whose scheduling algorithm emulates a stochastic chemical reaction process. Code and data may be rewritten during the reaction process, threads may fork and join, giving rise to a dynamic system in which code and data are constantly being modified. Indeed, PlasmidPL is heavily based on self-modifying code: due to the absence of explicit variables and external data structures, programs must rewrite themselves in order to get the right values in the right places where they are needed.

In this paper we present the syntax and behaviour of PlasmidPL programs, and show how they can be used to produce code that rewrites itself in order to implement steps from evolution runs such as initial population generation and genetic operators. We then show how these elements can be used in a simple symbolic regression problem: although it looks like a fairly classical GP run, an important difference must be highlighted: the code generation and modification operators act on the same program (or plasmid "soup") where they are located. This is a first step towards evolving self-modifying programs and their genetic operators as done in [7,8], this time using a chemical metaphor closer to nature.

After a brief introduction to biological plasmids in Sect. 2 and some literature review in Sect. 3, the language is described in Sect. 4. First GP experiments are reported Sect. 5, and further steps, insights and perspectives in Sect. 6.

2 Biological Plasmids

A *plasmid* is a small DNA molecule that can exist and replicate separately from the chromosomal DNA. It is typically circular and double-stranded. Plasmids are most commonly found in bacteria, but have also been found in eukaryotes. Fig. 1 (left) schematically depicts a set of plasmids inside a bacterium.

The structure of a plasmid comprises its *ori (origin of replication)* region, and a set of genes (Fig. 1 (centre)). The ori region is a nucleotide sequence that unites the two extremities of the DNA. During replication, this region is nicked and the DNA is duplicated starting from there. When the replication process is complete the plasmid recircularizes.

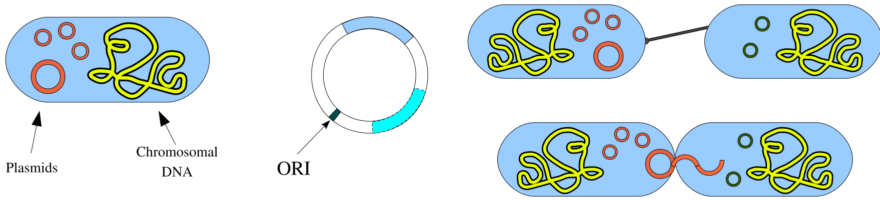


Fig. 1. Left: Bacterium holding chromosomal DNA and plasmids. Centre: A plasmid with its genes (large segments) and its origin of replication (*ori*). Right: A plasmid being transferred from a donor bacterium to a recipient during conjugation.

Plasmids are mobile genetic elements: they often migrate from one bacterium to another via *conjugation*, an asexual mechanism by which genetic material is transferred from a donor cell to a recipient by direct contact (Fig. 1 (right)).

Some plasmid genes may confer selective advantages to the host bacterium, such as antibiotic resistance. Since plasmids may reproduce and migrate to other bacteria, they are important transmission vectors of antibiotic resistance. Plasmids are also extensively used as cloning vectors in genetic engineering. More recently, *Plasmid Computing* has been proposed as a new DNA computing technique that uses plasmids to store information, capable of solving NP-complete problems such as the maximal independent subset problem [9], and the knapsack problem [10].

3 Background and Related Work

In most GP systems the code to be optimized is produced and manipulated by an external program. We would like to explore the possibility of including all the evolutionary steps within the code itself, like in nature, where DNA reproduces with the help of proteins that are manufactured from genes within the DNA itself. Systems with these characteristics have been extensively studied in Artificial Life and Complex Systems research [11,4,12]. Many such systems exhibit interesting life-like properties such as self-reproduction [12], self-evolution [6], self-maintenance [11,4], and so on. They are typically meant to study biological or organizational issues in general, such as the origin of life and the emergence of self-organizing structures. Artificial Chemistries [4] are intimately related to evolution: it is conjectured that evolution itself could have emerged out of catalytic chemical reactions. Modelling programs with an artificial chemistry could perhaps show evolutionary paths that remain unexplored so far.

Artificial Polymer Chemistries [4] are artificial chemistries in which molecules are character sequences that can be concatenated or cleaved during the reactions. PlasmidPL can be seen as a instance of an artificial polymer chemistry, in which molecules contain code that can be transformed via GP. However, our goal is not to study life phenomena but to optimize programs that offer services to users.

The notion of active code strands that operate on passive data strands is present in several earlier artificial chemistries [13,11,6,14]. Similarly, earlier

ring-based computation models exist. For instance, in [14] machines operate on circular tapes that contain a searched pattern, resulting in a system that is able to evolve simple self-replication loops up to complex autocatalytic networks. The potential benefit of PlasmidPL over such systems is to offer a fully blown and intuitive programming language based on a chemical metaphor, in which programs can be produced by humans as well as automatically via GP. Such hybrid approach has been successful in mainstream GP and could bring the chemical metaphor closer to everyday applications.

Our approach share goals with *Ontogenetic Programming* [8] and *Autoconstructive Evolution* [7], both based on variants of the Push language [15]. We took inspiration from Push for some aspects of PlasmidPL, which will become evident in Sect. 4.2. Languages such as MGS [16] are also inspired by a chemical metaphor. However, they have not been designed for GP, and as such, their syntax does not lend itself to easy automatic manipulation.

4 PlasmidPL: Language Description

As in most chemical models, PlasmidPL models programs as a multiset of virtual molecules (a set in which elements may occur more than once). Virtual molecules are rings or plasmids, represented as vectors of indivisible atoms or symbols. All reactions are second-order, i.e. involve two reactants. At each iteration, the reactor selects two molecules at random for reaction: the first one is chosen among the currently active (executable) molecules, and the second among the passive (data) ones.

The symbols in a plasmid are indexed starting from zero at the ori junction point. Any integer index $i \in \mathbb{Z}$ is converted to an index j, $0 \leq j < L$ which always falls within a valid position in the plasmid:

$$
\begin{array}{ll}
j = i\%L & \text{if } i \geq 0 \\
j = (L - (|i|\%L))\%L & \text{if } i < 0
\end{array}
$$

where L is the length of the plasmid in number of atoms and % is the modulo (division remainder) operator. Empty plasmids with $L = 0$ do not make sense and are never present in the multiset, therefore division by zero does not occur. For protected GP, floating point indices are rounded to the nearest integer before the above computation, and other (non-numeric) atoms are treated as zero.

Logically a plasmid wraps around itself in a circular shape. Physically however (in the source code), a plasmid is represented simply as a list of atoms in textual form (a_0 a_1 ... a_{L-1}) equivalent to a LISP list without recursion. The ori position is not represented, and is logically situated between atoms a_{L-1} and a_0. The list format also allows us to refer to plasmid positions informally as *front* positions (near the head of the list) and *rear* positions (near the tail's end).

For computational efficiency in treating long polymers, molecules are identified by *keys* stored in their front symbols. Any atom may in principle play the role of a key. Keys are used to identify reactants, as will be explained next.

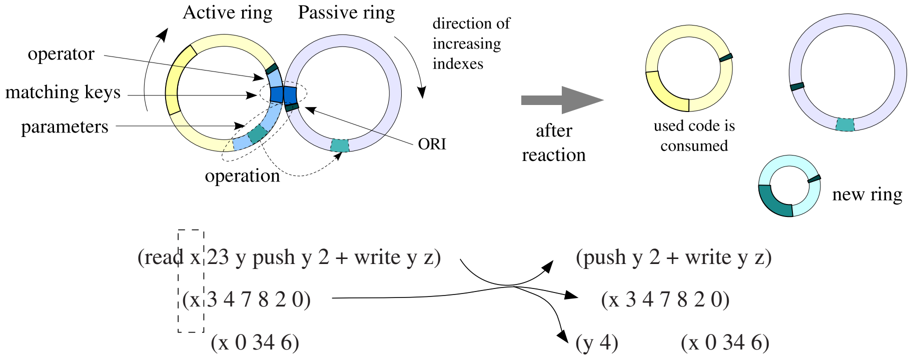


Fig. 2. Reaction between two plasmids. Top: Schematic representation. Bottom: Code example. An active (code) plasmid operates on a passive (data) plasmid.

Fig. 2 depicts the plasmid reaction metaphor. An active and a passive plasmid react if they have the same key (`x` in the example) at their index positions one and zero, respectively. The active plasmid operates on the passive one to obtain reaction products. Both plasmids may be modified in the process, while the executed code of the first (active) plasmid is consumed. The front keyword of the active molecule (i.e. the atom at position zero, `read` in the example) determines the action to be performed on the passive one, according to Reaction Table 1. The action may result in a modification of the passive molecule, of the active molecule itself, and/or the production of another molecule. After the action is performed, the atoms corresponding to the executed code fragment in the active molecule are consumed. In the example of Fig. 2, a `read` instruction produces a third plasmid `(y 4)` containing the read value. `y` is the output key given as a parameter to the `read` instruction, and `4` is the atom at position $23\%L = 2$ of molecule `(x 3 4 ...)` with length $L = 7$. Note from the bottom code example that two passive plasmids with the same key `x` occur: they have equal chance to participate in the reaction, and the first one is chosen at random.

The virtual chemical reactor is driven by a variant of the Gillespie algorithm [17], which simulates the stochastics of a real tank reactor by determining which reactions should occur and when. The choice of reactants is based on the molecule key. This partitions the set of all molecule chains in the reaction into species groups identified by the same key symbol. All the molecules belonging to the same group are treated as the same molecular species as input to the Gillespie algorithm. The search for matching reactants can then be reduced to a simple key lookup in a hash table. Multiple copies of identical rings are represented only once in the reactor by incrementing the *multiplicity counter* (number of copies of an item in the multiset) of the molecule. The resulting algorithm has complexity $\mathcal{O}(s)$ per iteration, where s is the total number of distinct front key symbols.

4.1 PlasmidPL Reactions

The language instructions specify chemical reactions that operate on a molecule whose front atom matches the reaction key. The syntax and semantics of the reaction rules are shown in Table 1.

Table 1. Semantics of PlasmidPL reactions

`read R P S`	read symbol at position `P` from ring `R`; result goes to ring/position `S`
`write R P S`	write symbol `S` at position `P` on ring `R`
`insert R P S`	insert symbol `S` into ring `R` at pos `P`
`delete R P`	delete symbol at position `P` from ring `R`
`join R`	join ring `R` with current ring (append)
`cleave R P`	divide ring `R` between ori and position/character `P`
`copy R`	duplicate ring `R`
`destroy R`	destroy ring `R`
`length R P`	length of ring `R` (written to ring/position `P`)
`exec R`	spawn ring `R` for execution; equivalent to `(copy R delete R 0)`
`push R S`	push symbol `S` onto stack `R`; if `S` is an operator, pop items from `S`, perform the operation and push the result onto `R`
`pop R P`	pops top of stack `R` and writes its value in ring/position `P`
`if C... else...`	if top of stack `C` is true, consume region after `else` atom, else consume region before `else`

The general syntax of a code fragment corresponding to a reaction instruction is (*keyword* k p_1 p_2 ..) where *keyword* is the reserved atom that indicates the type of reaction to be performed (e.g. `read`, `write`), k is the matching key that indicates which molecules may be chosen as passive side for the reaction, and p_i is a set of parameters. Parameters may indicate a position in the passive ring where information should be read/written from/to, the key for a new ring to be produced as output, the specific symbol value to be written to a ring, and so on. As a result of the reaction, rings may join, divide, duplicate, disappear, change from active to passive and vice-versa, become shorter or longer, etc.

For conciseness, we abuse the notation as follows: "ring `R`" means "a ring with atom `R` at position zero", i.e. a ring of the form `(R ...)`. "Ring/position X" refers either to a new ring `(X ...)` (if `X` is a non-numeric atom) or to an offset `X` (if `X` is a numeric value) at the self-molecule (the current active molecule) after the current code fragment (which will be consumed after the reaction). This apparently awkward syntax is intended both to facilitate code rewriting and to ensure that all possible parameters and their types are accepted as valid.

The `read` instruction is a typical example of this double semantics:

```
(read a 1 y), (a 10) → (y 10)
(read a 1 3 write z 1 V), (a 10) → (write z 1 10)
```

In the first case `(read a 1 y)`, a new molecule `(y 10)` is produced containing the value read from `(a 10)`. In the second case `(read a 1 3)` the same value is

written to the self molecule, which can then be used as a parameter for the next instruction in the flow of execution.

An exhaustive description of the instruction set is outside the scope of this paper. Sect. 5 will show some concrete code examples that will hopefully highlight the basic language principles in a practical way.

4.2 Arithmetic and Logic Expressions

We recall that rings are generic data structures that may contain code and data. As such, they can be used to store multiple data structures such as lists, vectors or stacks. Arithmetic and logic expression can be more easily evaluated by looking at a ring as a stack. Three instructions currently have such a semantic: `push`, `pop`, and `if`. This section focuses on the `push` instruction which will be used in the experiments of Sec. 5.3.

When reacting with a matching ring `R`, the `push` instruction uses this ring as a stack: the front portion is the top of the stack and the rear is the bottom. It works as follows: the atom `S` in the self-molecule immediately after the key `R` is inspected: If `S` is an operand (e.g. number) push it to ring `R`. If it is an operator (e.g. `+ - *`), pop needed operands from stack, perform operation and push the result onto `R`. In both cases the corresponding atom is consumed from the active ring. A reaction keyword in the place of `S` (or an empty `S`) acts as a stop condition: the `push` keyword is consumed, together with its key `R`. The stack `R` now contains the result of the computation.

The `push` reaction evaluates a postfix expression like a conventional stack-based language. However the evaluation does not happen at once, but one atom at a time, each time the molecule is chosen for reaction. This preserves the molecule thread model in the sense that other molecules may be chosen for processing in between. The following example shows the reaction path (execution trace) for incrementing a counter `c`:

```
(push c 1 +), (c 4) → (push c +), (c 1 4) → (push c), (c 5) → (c 5)
```

If there were more instances of `c` molecules in the soup, each instance would have an equal chance to react with `(push c ..)`, regardless of their content, leading to a non-deterministic execution which is very characteristic of chemical models.

The `push` reaction accepts arithmetic, logic and comparison operators which look like those in the C language, plus stack manipulation operators and a number of mathematical functions, including a random number generator. The set of operators can be easily enhanced. The current set is listed in Table 2.

For protected GP operations, all the stack operators follow the principle adopted by the Push language [15]: when insufficient arguments are available on top of the stack, or when arguments do not match the required types, or when arguments do not satisfy the pre-conditions for an operation (for example, in case of division by zero) then the operator is interpreted as a *nop* (no operation) and simply discarded without touching the stack.

Table 2. `push` reaction operators

`+ - * / % ^`	arithmetic operators (plus, minus,... mod, power)
`== != > < >= <=`	comparison (equal, not equal, greater than...)
`&& || !`	logic operators (and, or, not)
`swap, dup, del`	stack manipulation: swap the two symbols on top of stack, duplicate or delete symbol on top of stack
`sqrt, log, sin, cos`...	various math functions (square root, logarithm...)
`rnd, int.rnd`	random number generation (float and integer, respectively)

5 Genetic Programming with PlasmidPL

In this section we show hand-made examples of PlasmidPL programs that implement initial program generation, mutation and crossover in GP. In contrast to most GP systems, these GP operations are performed within the program itself that is being evolved. This is a first step towards evolving code that self-modifies, within which the genetic operators would naturally co-evolve.

5.1 Generating an Initial Program

Here we show a PlasmidPL program that generates a random postfix expression by composing it from a pool of atoms.

```
(genexpr push len dup 0 >
   if len
      push len 1 -
      read atom 1 3
         insert templ 3 A
      exec genexpr
   else
      write templ 0 expr
      destroy len)
```

```
(templ push stack
   write stack 0 result)
(push maxlen dup 1 - int.rnd 1 +
   read maxlen 1 len
   exec genexpr)
(maxlen 6)
(atom 0)  (atom 1) ...
(atom +)  (atom *) ...
(atom dup) ...
```

The left side shows the `genexpr` molecule, which upon activation by an `exec` ring generates an initial expression by combining random atoms taken from the pool of `atom` molecules on the right side. An initialization phase occurs first: the pair `(push maxlen )` and `(maxlen )` react together to produce a molecule `len` containing the target length of the expression to be generated, chosen randomly between one and the maximum length indicated within `maxlen` (6 atoms in this example). After that, `genexpr` is executed: if the length is positive, it decrements it, reads one atom at random from the `atom` pool, and inserts the atom within the template molecule `temp` between the `push stack` and `write` atoms. Then it invokes `genexpr` again, until the target length reaches zero. The template `templ` is then renamed to `expr`, and the temporary `len` molecule is destroyed. `expr` now contains the final expression that will be later evaluated. A sample expression generated in this way is:

```
(expr push stack + 4 5 + / - write stack 0 result)
```

Note that save the fixed prefix and suffix atoms from the template, the expression itself is a random sequence of atoms obeying no syntactic rules. It can nevertheless be evaluated since all the operators are protected.

The same principle could be applied to generate other types of programs, by injecting different `atom` molecules containing the symbols to be composed, e.g. `(atom read)`, `(atom write)`, etc. In this case, however, in order to improve the feasibility of the resulting programs, the modified `genexpr` molecule would have to contain code that inspects the chosen atom and fills in the next atoms with the expected number of parameters. This would be a next step in showing how entirely new code can be produced by rewriting rings.

5.2 Genetic Operators

Code extracts for mutation and crossover are shown below. They operate on the expression `expr` generated as described in Sect. 5.1.

Each `mutate` molecule implements a different type of mutation. For conciseness the choice of the mutation point is omitted, and only the code fragment that actually performs the mutation is shown: the first molecule replaces one symbol at random in the expression, using a `write` to a position `P` which is previously rewritten with the actual index. The value `V` to be written is also previously rewritten by the `read atom 1 3` instruction, which reads one atom at random from those listed in the example of Sect. 5.1. The other `mutate` molecules insert or delete one symbol at random, respectively, in a similar way.

An `(exec mutate)` molecule might react equally likely with each of the `mutate` variants (atom replacement, insertion, and deletion, respectively). The probabilities of each variant may be adjusted by injecting multiple copies of each in different proportions into the reactor.

```
(mutate ...
   read atom 1 3
   write expr P V ...)
(mutate .. insert expr P V ..)
(mutate .. delete expr P ..)
```

```
(crossover
   .. insert cp11 cp12 P
   .. insert cp11 |    P
   cleave cp11 | exec mk1 ..)
(mk1 .. join cp11 join cp22 ch1)
```

The right box above shows fragments of a simple one-point crossover operator. It takes two input expressions, both in the format shown in Sect. 5.1. one of them is the native expression that already resided in the reactor, coming from the initial population generation step. The second one is a copy of another plasmid's expression, injected via a conjugation mechanism. After choosing a crossover point at random in both expressions (which will be written into atom `P`'s position) an identifier for the second segment (`cp12` in the above example) and a breakpoint atom '`|`' are inserted at the crossover point. The rings are then broken up at the breakpoint position and recombined in a crossed way. The result of the crossover is available in molecules `ch1` and `ch2` (not shown), which are later captured for building a new generation as will be shown in Sect. 5.3.

Here is a simplified reaction trace of the crossover mechanism:

```
(cp11 a b c d), (cp21 x y z) →
(cp11 a b | cp12 c d), (cp21 x | cp22 y z) →
(cp11 a b), (cp12 c d), (cp21 x), (cp22 y z) →
(ch1 a b y z), (ch2 x c d)
```

5.3 Postfix Symbolic Regression

The expression generator from Sect. 5.1 together with the genetic operators from Sect. 5.2 can be combined to solve a simple symbolic regression problem. Each individual in the population is a chemical reactor that exchanges molecules with the external environment. The result is a fairly standard stack-based GP run, except that the code for the generation and modification of individuals is included within the individuals themselves. Besides such code, each individual contains code that: *(i)* obtains an input value corresponding to a fitness case and pushes it onto the stack molecule used to evaluate the automatically generated expression; *(ii)* when the computation is finished, rewrites the stack into an output molecule that is going to be read by an external fitness evaluator. Fitness evaluation and selection are implemented externally, simulating the fact that individuals must survive some environmental pressure. The target function is the well-known quartic polynomial. The fitness function is the sum of the squares of the errors between the obtained and the expected values.

Table 3. Koza tableau of symbolic regression experiments

objective:	evolution of a quartic polynomial in postfix form $f(x) = x + x^2 + x^3 + x^4$	crossover prob.:	90%
		mutation prob.:	5%
		selection:	tournament size 4
atom set:	`+ - * / ^ dup swap` `0 1 2 3 4 5`	termination criterion:	zero fitness
		max. # generations:	100
fitness cases:	5 values $x \in [-2; 2]$	max. program size:	none
pop. size:	100 individuals	initialization method:	grow

The Koza tableau for the experiments is shown in Table 3. The population size, maximum number of generations, number of fitness cases and tournament size are intentionally kept small. For simplification no ephemeral constants are used. Moreover, since there are no explicit variables, the program is obliged to take all its input at the beginning when it can be found on the top of the stack.

Fig. 3 shows the evolution results averaged over 10 runs. The average fitness of the best individual decreases with the generations, as expected. Program sizes grow visibly, as well as the variation in sizes. This can be expected as naive mutation and crossover operators were used, with no intron growth control.

One of the runs finds the following 100% correct solution at generation 60:

```
(swap dup dup dup * + swap dup dup dup * + swap 2 ^ * +)
```

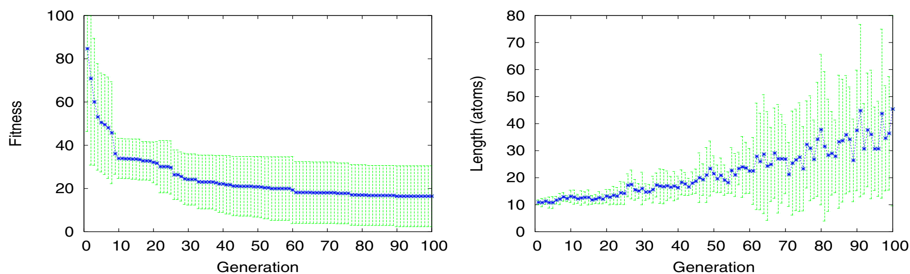


Fig. 3. Evolution of a quartic polynomial using PlasmidPL. Left: average fitness of the best individual of each generation. Right: average length of best of generation.

It evaluates exactly to: $f(x) = (x + xx) + (x + xx)(x^2) = x + x^2 + x^3 + x^4$ which is the expected expression. It leaves no garbage on the stack and contains only one "intron" (the first swap instruction).

Hand-made solutions include:

```
(dup dup dup 1 + * 1 + * 1 + *)
(dup 4 ^ swap dup 3 ^ swap dup 2 ^ swap + + +)
```

The solution found by GP is parsimonious and at a comparable level of efficiency to hand-made ones. This is remarkable since neither intron growth mechanisms nor stack-correct genetic operators were used.

We have also performed experiments with a quintic polynomial and other functions, with similar results (omitted for conciseness).

6 Discussion

We have described PlasmidPL and shown some examples of how it can be used for a form of GP based on code rewriting. First experiments show that it is feasible to perform fairly "standard" GP with PlasmidPL. This is not the main goal of the language, but only the very first steps. The goal is to have a system that can make use of a chemical metaphor that includes self-modification to control code evolution intrinsically. A research issue in this context is the trade-off between the stochastic nature of program execution, and the potential robustness that could achieved through the redundancy provided by multiple molecules performing similar functions. As pointed out in [5], in an algorithmic chemistry the concentration of instructions is more important than their execution order. A robust online evolution scheme would thus inherently rely on the control of code concentrations.

Experiments with more complex target programs must be performed next. Although helpful, the concept of rings is not sufficient to ensure viable individuals. At the current stage, most instructions expect a fixed number of parameters. This is prone to a *frame shift* in case of a random mutation: a missing or extra parameter may shift the whole execution to a different point, having the effect of

a macro-mutation. This can be fixed by allowing a variable number of parameters, combined with protected variation operators that operate on frame borders instead of random positions.

Many aspects of state-of-the-art GP have not been treated here: recursion, modularity, data types, and so on. These should in principle also be possible with a chemical model, however the requirement to operate exclusively on molecules would certainly impose new approaches to these problems.

7 Conclusions and Future Work

We presented PlasmidPL, a new programming language inspired by circular DNA structures called plasmids. The language is still in early stage of development, and is currently only loosely based on real biological plasmids. This paper shows some initial feasibility experiments. PlasmidPL is intended for online tasks related to environment monitoring and control, such as reacting to chemicals present in the environment by diffusing other chemicals in a controlled way.

The chemical metaphor offers a more biologically plausible model of evolution with the potential of autonomous evolution without external support. Indeed, Fontana [11] pointed out that in the physical universe the level of molecules is the only one that "has been observed to spontaneously support complex phenomena as life". In his chemistry based on λ-calculus, molecules (λ-expressions) represent code and data that operate on each other in a standalone way, thus do not really on a centralized machine architecture. We seek to bring these benefits to practical applications in the GP context. By moving a step closer to chemistry, the building blocks of life, new behaviours might emerge from evolved programs.

Acknowledgments

This work has been partially supported by the European Union through IST FET Project BIONETS (http://www.bionets.eu). The author thanks Christian Tschudin, Thomas Meyer, and Daniele Miorandi for their helpful comments.

References

1. Ziegler, J., Banzhaf, W.: Evolving Control Metabolisms for a Robot. Artificial Life 7(2), 171–190 (2001)
2. Taylor, T., Ottery, P., Hallam, J.: An approach to time- and space-differentiated pattern formation in multi-robot systems. In: Proc. TAROS (2007)
3. Dressler, F., et al.: Efficient Operation in Sensor and Actor Networks Inspired by Cellular Signaling Cascades. In: Proc. Autonomics, Rome, Italy (2007)
4. Dittrich, P., Ziegler, J., Banzhaf, W.: Artificial Chemistries – A Review. Artificial Life 7(3), 225–275 (2001)
5. Banzhaf, W., Lasarczyk, C.: Genetic Programming of an Algorithmic Chemistry. In: GPTP II, O., et al. (eds.), vol. 8, pp. 175–190. Kluwer/Springer (2004)

6. Dittrich, P., Banzhaf, W.: Self-Evolution in a Constructive Binary String System. Artificial Life 4(2), 203–220 (1998)
7. Spector, L., Robinson, A.: Genetic Programming and Autoconstructive Evolution with the Push Programming Language. GPEM Journal 3(1), 7–40 (2002)
8. Spector, L., Stoffel, K.: Ontogenetic programming. In: Proc. Genetic Programming 1st Annual Conf., Stanford University, CA, USA, pp. 394–399. MIT Press, Cambridge (1996)
9. Head, T., et al.: Computing with DNA by operating on plasmids. BioSystems 57, 87–93 (2000)
10. Henkel et al., C.V.: DNA computing of solutions to knapsack problems. BioSystems 88(1-2), 156–162 (2007)
11. Fontana, W., Buss, L.W.: The Arrival of the Fittest: Toward a Theory of Biological Organization. Bulletin of Mathematical Biology 56, 1–64 (1994)
12. Sipper, M.: Fifty years of research on self-replication: an overview. Artificial Life 4(3), 237–257 (1998)
13. Laing, R.: Automaton models of reproduction by self-inspection. Journal of Theoretical Biology 66, 437–456 (1977)
14. Ikegami, T.: Evolvability of Machines and Tapes. Artificial Life and Robotics 3(4), 242–245 (1999)
15. Spector, L., Klein, J., Keijzer, M.: The Push3 execution stack and the evolution of control. In: Proc. GECCO 2005, Washington DC, USA, pp. 1689–1696 (2005)
16. Giavitto, J.-L., Michel, O.: MGS: a rule-based programming language for complex objects and collections. Electr. Notes in Theor. Computer Science 59 (2001)
17. Gillespie, D.T.: Exact Stochastic Simulation of Coupled Chemical Reactions. Journal of Physical Chemistry 81(25), 2340–2361 (1977)

Using Genetic Programming for Turing Machine Induction

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. Turing machines are playing an increasingly significant role in Computer Science domains such as bioinformatics. Instead of directly formulating a solution to a problem, a Turing machine which produces a solution algorithm is generated. The original problem is reduced to that of inducing an acceptor for a recursively enumerable language or a Turing machine transducer. This paper reports on a genetic programming system implemented to evolve Turing machine acceptors and transducers. Each element of the population is represented as a directed graph and graph crossover, mutation and reproduction are used to evolve each generation. The paper also presents a set of five acceptor and five transducer benchmark problems which can be used to test and compare different methodologies for generating Turing machines. The genetic programming system implemented evolved general solutions for all ten problems.

Keywords: Turing machines, genetic programming, grammatical inference.

1 Introduction

As the potential of Turing machines in more recent domains of Computer Science such as bioinformatics is being realized, interest in the automatic induction of these automata is growing. While there has been a number of investigations into the evolution of other forms of automata such as finite acceptors ([2] and [4]) and transducers ([3] and [6]), there has not been much research into the use of evolutionary algorithms for Turing machine induction. Research in this domain was initiated by the study conducted by Tanomaru [7] in the early nineties. The main motivation of Tanomaru's study was that once a Turing machine is evolved it can be easily implemented as a computer program solution to the problem. Tanomaru used an evolutionary algorithm to generate Turing machine transducers for two problems. This system did not scale well when applied to Turing machine acceptor problems and "population shifting" was introduced to overcome this problem. The revised system evolved solutions to three acceptor problems.

Since this initial study there has not been much research into this domain. In the late nineties Pereria et al. [5] have evolved solutions to the Busy Beaver problem. The main aim of this study was to test the effectiveness of using graph-crossover instead of

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 350–361, 2008.
© Springer-Verlag Berlin Heidelberg 2008

two-point crossover. Graph-crossover was found to improve the success rate of the evolutionary algorithm and the system produced some of the best results in this field. In 2001 Vallejo et al. [8] implemented an evolutionary algorithm to induce Turing machine acceptors for recognizing HIV biosequences. The biosequences to be recognized are treated as a recursively enumerable language. The system found solutions that generalized well and correctly classified sequences not in the training set. The system was also successfully applied to evolving a two-way deterministic finite automaton which was used to find a solution to the multiple sequence alignment problem.

The study presented in this paper extends the research conducted by Tanomaru by evaluating the use of evolutionary algorithms for Turing machine induction on a larger set of problems. Furthermore, based on the success that Pereria et al. had with representing each Turing machine as a graph, a transition graph representation is used, rather than the state transition table representation used by Tanomaru.

The following section describes the genetic programming system implemented to evolve Turing machines and section 3 specifies the experimental setup used in applying the system to ten Turing machine problems. The performance of the system on ten problems is discussed in section 4. Section 5 summarizes the conclusions of this study and describes future extensions of this project.

2 The Genetic Programming System

This section provides an overview of the genetic programming system implemented to evolve Turing machines. As is the case of the three previous studies in this domain, the generational control model is used. The representation, method of initial population generation, methods for selection and fitness evaluation and the genetic operators are described in the sections that follow.

2.1 Representation and Initial Population Generation

Tanomaru [7] emphasizes the importance of representing Turing machines as "intuitively" as possible. He states that this facilitates easy and efficient translation of the Turing machine without loss of information. Thus, in the study conducted by Tanomaru each Turing machine is represented using a state transition table instead of a chromosome or a tree. Pereira et al. [5] take this idea a step further by representing each machine directly as a transition graph. Given the success that Pereira et al. had with this representation, it was decided to use the same representation in this study. In the previous three studies each Turing machine was of a fixed size. In order to allow for more of the search space to be explored it was decided that Turing machines would be of variable size in this study. However, an upper bound is set on the Turing machine size.

Each machine has access to two tapes, an input tape and an output tape. It is assumed that the input string is on the input tape and the read-write head is at the first character of this string. The tape is infinite on both sides and the cells not occupied by the input string contain the blank symbol (B). Note that the input tape is a read-only tape. In the case of Turing machine transducers, the output string must be written to the output tape. The system caters for two types of transducers. In the first type all contiguous non-blank output on the tape is considered to be the output of the machine and the read-write head does not necessarily point at the first character of the output string. In the second type

the read-write head is positioned at the beginning of the output on the output tape and the first blank encountered from this point onward is treated as the end of string marker. Initially the output tape contains blank symbols and the read-write head is pointing at any cell. The read-write head can move left (L), right (R), or remain where it is (S). **Figure 1** illustrates an example of an input and output tape before processing. The read-write head is represented as an arrow. The input string is *aba*.

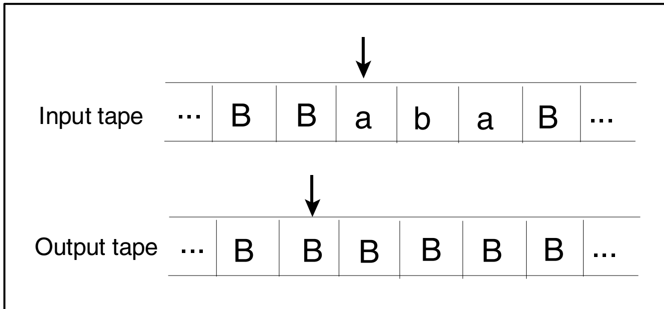


Fig. 1. Input and output tapes for a Turing machine

An example of a Turing machine transducer is illustrated in Figure 2 and a Turing machine acceptor is depicted in Figure 3. Note that each machine has a single *HALT* state. Each transition is defined as a two-tuple. The first component of the transition specifies the symbol that must be read from the input tape and the action of the read-write head for this tape. The second component of the transition describes the character that must be read from the output tape, the character to be written to the output tape and the action of the read-write head. The second component does not have to write a character to the output tape in which case the second argument is not specified. For example, *b/R, B/a/R* means that a *b* must be read from the input tape and the read-write head must move right; a blank is read from the output tape, an *a* is written to the output tape overwriting the blank and the read-write head is moved right for this tape. In order for a transition to be executed the character to be read in the first component must be on the input tape and the character to be read in the second component must be on the output tape.

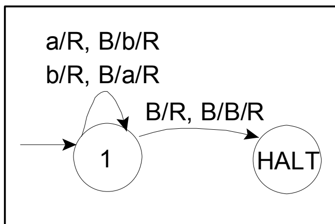


Fig. 2. Turing Machine Transducer

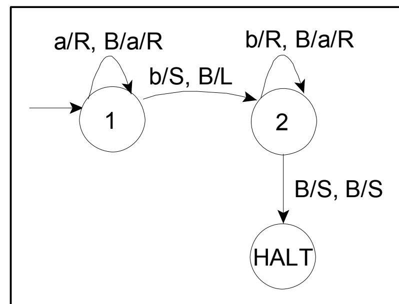


Fig. 3. Turing Machine Acceptor

Figure 4 illustrates how the input string *aba* is processed by the Turing machine in Figure 2. The first transition executed is *a/R, B/b/R* as there is an *a* on the input tape and a blank on the output tape. The read-write head on the input tape is now at the character *b* and the output tape read-write head is pointing at a blank. Thus, the next transition performed is *b/R, B/a/R*. This is followed by the application of the transition *a/R, B/b/R* again. Finally, the read-write heads on the input and output tapes are pointing at blanks which results in the transition *B/R,B/B/R* being executed causing the Turing machine to halt.

The process involved in creating a Turing machine is depicted in Figure 5. The first node created automatically becomes the start state. The arity of the node is randomly chosen to be in the range of 1 and the maximum number of transitions permitted. The start state cannot be a *HALT* state. Nodes representing successive states are randomly chosen to be a *HALT* state or not. Transitions connecting a node to each of its children are created by randomly choosing elements from the input alphabet, tape alphabet (in the case of the second component) and the read-write head actions. In the example in Figure 5 the third child of the start state has been randomly chosen to be the *HALT* state. If a *HALT* has not been chosen before the maximum number of nodes permitted is reached, the last state is designated as a *HALT* state.

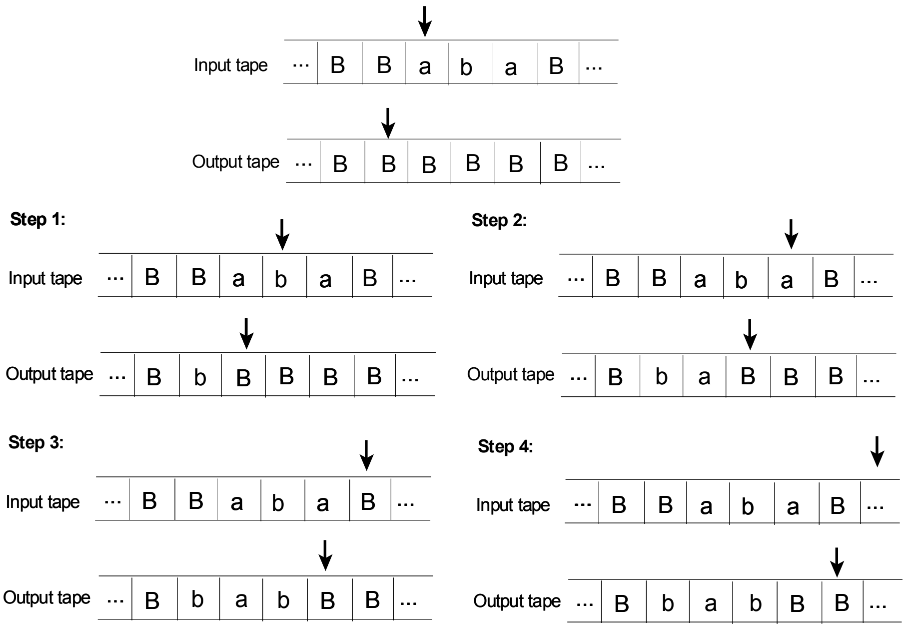


Fig. 4. Processing of the input string *aba* using the Turing machine transducer in Figure 2

Each element of the initial population is created in this way. This population is refined on successive iterations of the evolutionary algorithm. The following section describes methods used to calculate the fitness of an individual and select the parents of each generation.

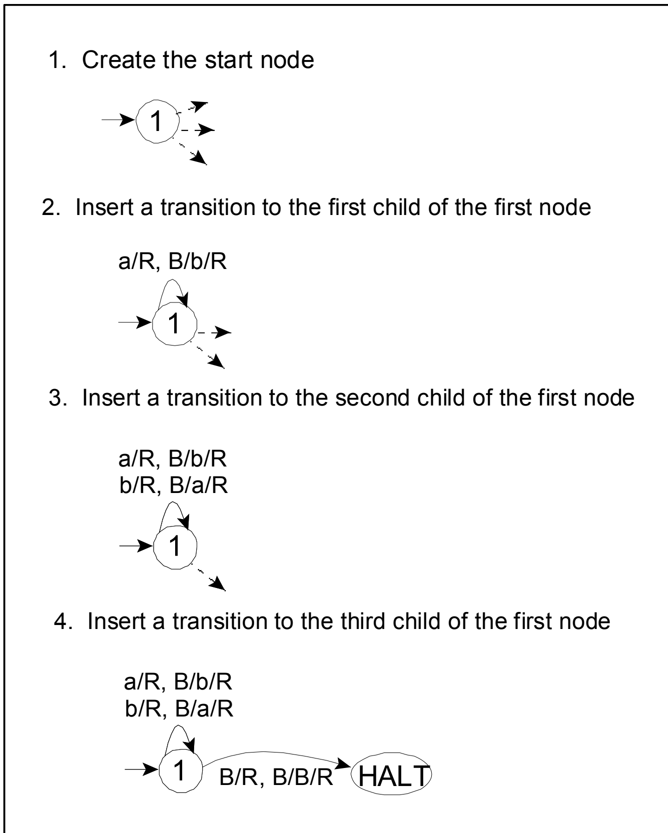


Fig. 5. Process of creating an element of the population

2.2 Fitness Calculation and Selection

The fitness of each element of the population is calculated by applying the individual to a set of fitness cases. The fitness cases for Turing machine transducers are comprised of pairs of input and the corresponding output strings. In the case of Turing machine acceptors the fitness cases are positive and negative sentences of the language, i.e. elements and non-elements of the language respectively.

Each input string is processed using the particular Turing machine and the output produced is compared to that of the corresponding target output. As in the study conducted by Pereira et al. [5] a limit is set on the number of transitions performed by a machine so as to prevent non-halting machines from running infinitely. If this limit is reached the machine is halted and the output at that point is compared to the target output for that fitness case. The machine is also halted if it reaches a *HALT* state or the current state does not have a transition for the combination of the characters currently being pointed to on the input and output tapes. An input string is accepted by an acceptor if the machine stops in a *HALT* state upon processing the entire input string. In all other cases the string is rejected.

The fitness of a Turing machine transducer is the number of fitness cases for which the machine produces the same output as that specified in the fitness case while the fitness of an acceptor is the number of sentences correctly classified as belonging or not belonging to the language. If the fitness of Turing machine is equal to the number of fitness cases, this machine is reported as a solution. These fitness values are used by the selection method to choose the parents of each generation. In this study tournament selection is used for this purpose. Selection is with replacement.

2.3 Genetic Operators

The reproduction, mutation and crossover operators are used to create the next generation. The parents chosen, using tournament selection, are passed to the genetic operators.

The mutation operator randomly chooses a mutation point. The sub-graph rooted at this position is removed. A newly created sub-graph is inserted at this point. The crossover operator is similar to the graph-crossover operator implemented by Pereira et al. [5]. Crossover points are randomly selected in each of the parents. The sub-graphs rooted at these points are swapped and the states are re-numbered in each of the offspring. HALT states cannot be chosen as crossover points.

The destructive effects of genetic operators often result in offspring with worse fitness than there parents and can lead to the GP system converging to an area of the search space in which a solution cannot be found [1]. Thus, if the offspring produced by mutation and crossover are not at as fit as the parents the operation is repeated until offspring with fitness at least as good as the parents are produced. This results in the algorithm converging quicker but could result in the system converging to local optima. To prevent this, a limit is set on the number of attempts at producing fitter offspring. If this limit is exceeded the current offspring is/are returned. Note that as a result of this the number of fitness evaluations cannot be directly determined from the number of generations used.

The following section describes the experimental setup used to test the genetic programming system on a set of ten Turing machine problems.

3 Experimental Setup

The GP system was tested on the five transducer and five acceptor problems listed in Table 1.

The GP system was implemented in Java (JDK version 1.5.0_06) and simulations were run on a Windows based 1.86 GHz PC with 1GB of RAM. Due to the randomness associated with evolutionary algorithms (not to prove statistical significance) ten runs, each using a different seed, were performed for each problem. Ten seeds were generated prior to the simulations and these ten seeds are used for all the problems. In order to prevent premature convergence of the system due to selection noise, multiple iterations were performed per seed. A maximum of ten iterations is permitted per seed for all problems. If a solution is found before the ten iteration limit is reached the process is terminated, the solution is reported and the run is counted as a successful run.

Table 1. Turing machine problems

Problem	Description
T1	A Turing machine to perform unary addition of two unary integers separated by a zero, e.g. if the input is *110111* the output string is *111110*
T2	A Turing machine that takes in a unary integer *n* and outputs *2n*, e.g. if the input string is *111* the output is *111111*
T3	A Turing machine that takes a unary integer *n* as input and outputs *2n+1*, e.g. if the input string is *11* the output is *11111*
T4	A Turing machine that takes two unary integers separated by zero as input and outputs the greater of the two integers, e.g. if the input is *1101* the output should be *11*
T5	A Turing machine to perform unary subtraction of two unary integers separated by a zero. If the second unary integer is larger or equal to the first unary integer a zero is output otherwise the difference, as a unary integer, is output, e.g. if the input string is *1111011* the output string is *111*
A1	$L=\{a^nb^n: n>=1\}$, $\Sigma=\{a,b\}$
A2	$L=\{awb: w \in \{a,b\} \}$, $\Sigma=\{a,b\}$
A3	$L=\{a^nb^na^n: n>=1\}$, $\Sigma=\{a,b\}$
A4	$L=\{ a^nb^nc^n: n>=1\}$, $\Sigma=\{a,b,c\}$
A5	$L=\{a^nb^{n+1}: n>=1\}$, $\Sigma=\{a,b\}$

Values for the GP parameters are listed in Table 2. These values were obtained empirically by performing trail runs of the system. The maximum size of machines during initial population generation is seven while the maximum size of offspring produced on successive generations is fifteen. If a genetic operator produces offspring exceeding this limit it is reapplied until an offspring of the correct size is produced. The evolutionary algorithm terminates if either a solution is found or the maximum number of generations has been completed. A Turing machine transducer is regarded as a solution if it produces the target output for all the fitness cases and a Turing machine acceptor is a solution if it correctly classifies all fitness cases.

Table 2. GP parameter values

Parameter	Value
Population size	2000
Maximum number of generations	100
Tournament size	5
Initial number of nodes	7
Maximum number of nodes	15
Application rates	Crossover: 50% Mutation: 40% Reproduction: 10%
Termination criteria	A solution is found or the maximum number of generations has been reached

A different number of fitness cases was used for each problem. These values are listed in Table 3 and were also obtained empirically by performing trial runs. The trial runs were used to identify an approximate number of fitness cases needed to sufficiently represent each problem domain so that the solutions evolved are not brittle. Thus, the number of fitness cases differs from one problem to the next.

Table 3. Number of fitness cases used for each problem

Problem	Number of Fitness Cases
T1	10
T2	3
T3	4
T4	22
T5	11
A1	405
A2	92
A3	106
A4	186
A5	276

Tanomaru [7] states that the lower success rates for the acceptors in his study could possibly be attributed to the low number of fitness cases used. Forty fitness cases were used for A1, A2 and A3 in the Tanomaru study. Thus, we have ensured that sufficient fitness cases have been provided for all acceptor problems in this study.

4 Results and Discussion

The GP system evolved general solutions, i.e. the solutions were not brittle and generalized well for the particular problem, to all ten of the problems listed in **Table 1**. An example of one of the transducers evolved for *T1* is illustrated in **Figure 8**.

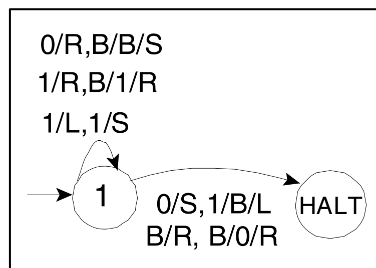


Fig. 6. One of the transducers evolved for *T1*

This transducer performs unary addition given an input string consisting of two unary integers separated by a zero. For example, if *11011* is the input string the transition *1/R, B/1/R* is the only transition satisfied at state *1* as the read-write head on the input tape is pointing at a *1* and the read-write head on the output tape is pointing at a *B*, i.e. a blank. This transition is applied twice at state *1*. The next transition applied is *0/R, B/B/S*. A *0* is read on the input tape and a blank on the output tape. The read-write head on the output tape remains in the same position while the read-write head on the input tape moves right and is now pointing at a *1*. The transition *1/R, B/1/R* is applied twice again resulting in the remaining *1*'s being written to the output tape and the read-write heads on both the

tapes pointing at blanks. This satisfies the transition *B/R, B/0/R* which writes a *0* to the output tape and halts the machine.

Note that the transition *1/L, 1/S* from state *1* to itself and the transition *0/S, 1/B/L* from the start state to the *HALT* state will never be executed. We refer to these as **structural redundancies**. A number of the evolved solutions for the ten problems were found to contain such redundant transitions. These redundancies increase the structural complexity of the evolved solutions and may indirectly increase processing time. Future work will examine these redundancies and the possible elimination of them in more detail.

Figure 9 depicts one of the acceptors induced by the system for *A1*. The language accepts all strings of the form a^nb^n with n>=1. Suppose that the input string is *ab*. The transition satisfied at the start state is *a/R, B/b/S* as the read-write head on the input string is pointing at the character *a* and the read-write head on the output string is pointing at a blank. Executing this transition results in the machine being moved to state *2* with the read-write heads on both the input and output tapes pointing at the character *b*. Thus, the transition *b/R, b/L* is executed. This results in the machine remaining at state *2* and both the read-write heads pointing at blanks. This satisfies the transition *B/L, B/B/L* at state *2* which causes the machine to halt. The machine has stopped at a *HALT* state, thus the string *ab* is accepted.

Alternatively, consider the input string *aab* which does not belong to the language. Again at the start state the transition *a/R, B/b/S* is executed resulting in the machine moving to state *2* with the second *a* being read in on the input tape and the read-write head of the output tape pointing to the *b* that has just been written to it. The transition *a/S, b/R* is then executed. The read-write on the input tape remains at the *a* and the read-write head on the output tape is now pointing at a blank. This transition results in the machine moving back to state *1* and the transition *a/R, B/b/S* being executed again followed by the transition *b/R, b/L* at state *2*. The read-write head of the input tape is pointing at a blank and the read-write head of the output tape is pointing at the character *b*. There is no transition at state *2* which satisfies this combination, thus the machine halts. The machine has halted in a state that is not a *HALT* state and the input string is therefore rejected.

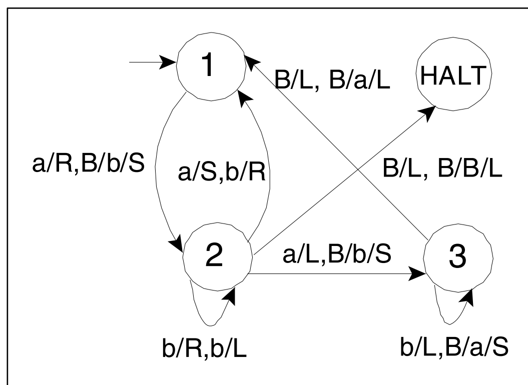


Fig. 7. An example of a solution evolved for *A1*

The success rates for the ten problems are listed in Table 4. The success rate is the percentage of the ten runs performed that produced a general solution. As in the study conducted by Tanomaru [7] the system did not experience any difficulties in evolving Turing machine transducers and a success rate of 100% was obtained for all five transducer problems. The problem of inducing acceptors is more complicated as the Turing machine has to accept a set of strings while at the same time reject all other string combinations for the given alphabet. In the case of a transducer the machine has to produce a single output string corresponding to the input string. Thus, the search space is larger for acceptors than transducers. For the acceptor problems *A1*, *A2* and *A5* the success rates are a 100% or close to 100%. The failure of the system to find a solution on one iteration for *A1* and *A5* can possibly be attributed to selection noise.

The success rates for *A3* and *A4* are much lower compared to that of the other acceptor problems. These problems are slightly more complicated than the other acceptor problems and have larger search spaces. Future work will investigate the low success rates in more detail. It is suspected that the system is not exploring a wide enough region of the search space. The use of similarity indexes will be introduced during initial population generation to ensure that the initial population represents more of the search space. The effects of the genetic operators will also be studied in detail and refined accordingly if necessary.

Table 4. Success rates for the ten Turing machine problems

Problem	Success Rate
T1	100%
T2	100%
T3	100%
T4	100%
T5	100%
A1	90%
A2	100%
A3	10%
A4	10%
A5	90%

The results obtained by the system for the acceptor problems, given the particular parameter values used, are comparative to that obtained in the Tanomaru study. The success rates obtained by the Tanomaru system are listed in Table 5. Note that the number of runs performed per problem in the Tanomaru study is a hundred while ten runs were performed in this study. The success rate for each problem in Table 5 is the percentage of the hundred runs that have produced solutions for the problem. The results obtained in this study appear to be an improvement over the previous results obtained for Turing machine acceptors, however a direct comparison is difficult as the Tanomaru system used population shifting while the system presented in the paper does not. Furthermore, the system presented in the paper uses multiple iterations to escape local optima caused by selection variance and the genetic operators implemented incorporate a form of hill-climbing as a preventative measure against the destructive effects of crossover and mutation. This improvement needs to be studied further to draw more concrete conclusions.

Table 5. Success rates for acceptor problems for the Tanomaru study

Problem	Without population shifting	With population shifting
A1	9%	82%
A2	41%	62%
A3	1%	38%

5 Conclusion and Future Work

The study presented in this paper forms part of a larger initiative aimed at examining the possibility of evolving a Turing machine that produces a solution to a problem, rather than evolving the solution algorithm itself. This study extends the work carried out by Tanomaru [7] to evaluate evolutionary algorithms as a means of inducing solutions to basic transducer and acceptor Turing machine problems. The study presented in this paper has revealed that evolutionary algorithms are effective at generating Turing machine transducers. The GP system implemented obtained a 100% success rate for all five transducer problems. The system was also able to evolve general solutions to all five acceptor problems. The success rates obtained for two of the more complicated acceptor problems was not high. We suspect that this can be attributed to not enough of the search space being explored during initial population generation and by the genetic operators. Future extensions of this project will examine this further. Some of the solutions evolved by the system were found to contain structural redundancies which could indirectly increase the processing time of the Turing machine. Future work will also investigate the existence and possible removal of such redundancies.

Acknowledgments. The authors would like to thank the reviewers for their helpful comments and suggestions. This material is based on work financially supported by the National Research Foundation (NRF) of South Africa.

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 Publishers, Inc., San Francisco (1998)
2. 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), pp. 351–358. IEEE Press, Los Alamitos (2003)
3. Lucas, S.M.: Evolving Finite State Transducers: Some Initial Explorations. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E.P.K., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 130–141. Springer, Heidelberg (2003)
4. Luke, S., Hamahashi, S., Kitano, H.: Genetic Programming. In: Banzhaf, W., Daida, J., Eiben, A.E., Garzan, M.H., Honavar, V., Jakiela, M., Smith, R.E. (eds.) Proceedings of the Genetic Programming and Evolutionary Computation Conference, Orlando, Florida, USA, vol. 2, pp. 1098–1105 (1999)

5. Pereria, F.B., Machado, P., Costa, E., Cardoso, A.: A Graph Based Crossover – A Case Study with the Busy Beaver Problem. In: Banzhaf, W., Daida, J. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, vol. 2, pp. 1149–1155. Morgan Kaufmann, San Francisco (1999)
6. Naidoo, A., Pillay, N.: The Induction of Finite Transducers Using Genetic Programming. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 371–380. Springer, Heidelberg (2007)
7. Tanomaru, J.: Evolving Turing Machines. In: Hao, J.K., Lutton, E., Ronald, E., Schoenauer, M., Snyers, D. (eds.) AE 1997. LNCS, vol. 1363, pp. 167–180. Springer, Heidelberg (1993)
8. Vallejo, E.E., Ramos, F.: Evolving Turing Machines for Biosequence Recognition and Analysis. In: Miller, J.R., Tomassini, M., Lanzi, P.L., Ryan, C., Tettamanzi, A.G.B. (eds.) EuroGP 2001. LNCS, vol. 2038, pp. 192–203. Springer, Heidelberg (2001)

Altering Search Rates of the Meta and Solution Grammars in the mGGA

Erik Hemberg, Michael O'Neill, and Anthony Brabazon

Natural Computing Research & Applications Group
University College Dublin, Ireland
erik.hemberg@ucd.ie, m.oneill@ucd.ie, anthony.brabazon@ucd.ie

Abstract. Adopting a meta-Grammar with Grammatical Evolution (GE) allows GE to evolve the grammar that it uses to specify the construction of a syntactically correct solution. The ability to evolve a grammar in the context of GE means that useful bias towards specific structures and solutions can be evolved during a run. This can lead to improved performance over the standard static grammar in terms of adapting to a dynamic environment and improved scalability to larger problem instances. This approach allows the evolution of modularity and reuse both on structural and symbol levels resulting in a compression of the representation of a solution. In this paper an analysis of altering the rate of sampling of the evolved solution grammars is undertaken. It is found that the majority of evolutionary search is currently focused on the generation of the solution grammars to such an extent that the candidate solutions are often hard-coded into them making the solution chromosome effectively redundant. This opens the door to future work in which we can explore how the search can be better balanced between the meta and solution grammars.

1 Introduction

This paper aims to investigate if the performance of a search using meta-grammars and Grammatical Evolution (GE) can be improved if the grammars sampled by the meta-grammar are explored more thoroughly. The ability to evolve a grammar in the context of GE means that useful bias towards specific structures and solutions can be evolved during a run. This can lead to improved performance over the standard static grammar both in terms of the ability to adapt to a dynamic environment and scalability [1,8]. Meta-grammar GE adopts two chromosomes, one is used to map the meta-grammar to a solution grammar. The second chromosome is used to map the solution grammar to a candidate solution. Earlier studies in meta-grammar GE have used the same rate of search on both of these chromosomes by adopting the same rate of mutation and crossover on each. In this study, two approaches to altering the rate of exploration of the solution grammars are examined. The first adopts implicit sampling using different rates of mutation on the evolved solution grammar versus the solutions sampled from the evolved solution grammar. The second approach explicitly generates more than one sample from each solution grammar. The paper is structured as

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 362–373, 2008.
© Springer-Verlag Berlin Heidelberg 2008

follows. First an overview of the meta-grammar approach to GE is presented and earlier research in this area is exposed in Sec. 2. Sec. 3 describes the two experiments undertaken and results obtained. In light of the results further analysis is described in Sec. 4, before finishing the paper in Sec. 5 with Conclusions and Future Work.

2 Meta Grammars in Grammatical Evolution

The grammar-based Genetic Programming approach upon which this study is based is the Grammatical Evolution by Grammatical Evolution algorithm [1], Which is based on the Grammatical Evolution algorithm [2,3]. 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. See [2,4] for further examples of what can be represented with grammars and [5] for an alternative approach to grammar evolution. By allowing an Evolutionary Algorithm to adapt its representation (here through the evolution of the grammar) it is possible to automatically incorporate biases into the search process. In this case we can allow the meta grammar Genetic Algorithm to evolve biases towards different useful code blocks of varying sizes.

In $(GE)^2$ the meta 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 meta 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. 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 GE. In Fig. 1 a meta grammar GE is shown.

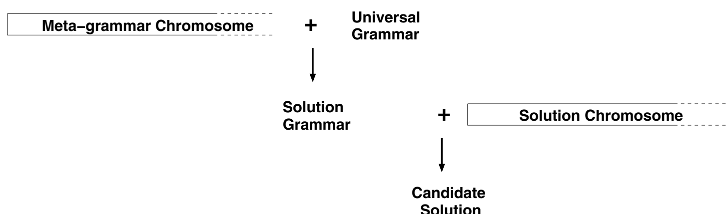


Fig. 1. An overview of the meta-grammar approach to GE

An example of a meta-grammar that could be used to evolve grammars for generating 8 bit binary strings is given below.

```
<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 bit string grammar that could be sampled from the above meta-grammar follows below. In this example, there are five possible forms that a `<bitstring>` can take on, with two possible choices for blocks of sizes 4, 2 and 1. The rule for generating a `<bit>` has four possible outcomes with a clear bias towards a `<bit>` becoming a `1` with a probability of 0.75.

```
<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 | 1 | 1
```

2.1 Earlier Research

There have been a number of studies of a meta-grammar approach to GE [1,6,7,8,9]. In each of these the same rate of evolutionary search was adopted on both the meta-grammar and solution chromosomes through the adoption of the same rates of mutation and crossover. The original study [1] investigated the feasibility of this approach and demonstrated its effectiveness in dynamic environments. In the mGGA [6] the meta-grammar approach was shown as an effective method to perform as an alternative binary string Genetic Algorithm through the provision of a mechanism to achieve modularity. A follow-up study demonstrated that the mGGA had an improved ability to scale to harder problem instances over the modular GA [10]. An observation of some of the solutions and solution grammars evolved by meta-grammar GE has exposed a tendency of generating grammars that did not have the possibility to produce many different strings [6,9].

3 Experiments and Results

In order to test the effects of more search on the solution grammar the following experiment was setup.

Checkerboard. In this problem a pattern of colours or states is imposed upon a two dimensional grid called the Checkerboard, introduced in [10]. There are 2 possible states adopted for each square on the grid, i.e., black or white, which here are represented as bit values 1 and 0 respectively. Fitness is simply measured by summing the number of squares that are in an incorrect state. Fig.2 which illustrates scaled-up versions of a pattern. The instances will be referred to by the total number of bits needed to describe the board. The board consists of 8 squares of consecutive zeros or ones. In this paper the board sizes used where $Cb_{32}, Cb_{72}, Cb_{128}, Cb_{200}, Cb_{288}$.

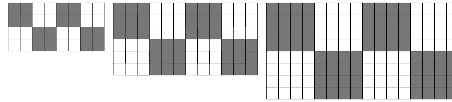


Fig. 2. Checkerboard patterns for Cb_{32}, Cb_{72} and Cb_{128}

Grammar. To solve the Checkerboard a meta grammar was created. To allow the creation of multiple blocks of different sizes the following grammar could be used. Below a is an example of a grammar for Cb_{32}.

```
<g> ::=  "<bitstring> ::=" <reps>
              "<bbk16> ::=" <bbk16>
              "<bbk8> ::=" <bbk8>
              "<bbk4> ::=" <bbk4>
              "<bbk2> ::=" <bbk2>
              "<bbk1> ::=" <bbk1>
               "<bit> ::=" <val>
<bbk16> ::= <bbk16t> | <bbk16t> "|" <bbk16>
<bbk8> ::= <bbk8t> | <bbk8t> "|" <bbk8>
<bbk4> ::= <bbk4t> | <bbk4t> "|" <bbk4>
<bbk2> ::= <bbk2t> | <bbk2t> "|" <bbk2>
<bbk1> ::= <bbk1t> | <bbk1t> "|" <bbk1>
<bbk16t> ::= <bit><bit><bit><bit><bit><bit><bit><bit><bit><bit><bit><bit><bit><bit><bit><bit>
<bbk8t> ::= <bit><bit><bit><bit><bit><bit><bit><bit>
<bbk4t> ::= <bit><bit><bit><bit>
<bbk2t> ::= <bit><bit>
<bbk1t> ::= <bit>
<reps> ::= <rept> | <rept>  "|" <reps>
<rept> ::= "<bbk16><bbk16>"  | "<bbk8><bbk8><bbk8><bbk8>"
         | "<bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4>"
         | "<bbk2><bbk2><bbk2><bbk2><bbk2><bbk2><bbk2><bbk2><bbk2><bbk2><bbk2><bbk2><bbk2>
            <bbk2><bbk2><bbk2>"
         | "<bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1>
            <bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1><bbk1>"
<bit> ::= "<bit>" | 1 | 0
<val> ::= <valt> | <valt> "|" <val>
<valt> ::= 1 | 0
```

When ss_i, the number of consecutive 1s or 0s, is a power of 3 a building block is created for 2^j where j is $0, 1, \ldots, n$ where $n = \log_2(N/2)$ and N is the total number of squares on the checkerboard. Otherwise ss_i the following is applied. For Cb_{72} the blocks are $32, 12, 6, 3, 1$, for Cb_{200} they are $100, 20, 10, 5, 1$ and for Cb_{288} they are $144, 24, 12, 6, 3, 1$.

3.1 Experiment 1 - Different Mutation Rates

The first experiment tests if there is any improvement in the performance of GE by having different mutation rates on the two chromosomes. The argument can be made that if the meta grammar should evolve at a slower pace then further exploration of the solution grammar would be possible, thus creating a version of a meta grammar local search.

Hypothesis. The performance is measured as the average number of fitness evaluations required for 30 runs to solve an instance of the Checkerboard. μ is the performance with the mutation being the same for both chromosomes. The performance for the chromosome having different mutation rates is referred to as μ_0. For each instance the following hypothesis is stated:

H_0**:** Equal mutation rate on both chromosomes has the same performance as or better performance then a lower mutation rate on the first chromosome i.e. $\mu \leq \mu_0$
H_1**:** Equal mutation rates have worse performance compared to a low mutation rate on the first chromosome i.e. $\mu > \mu_0$
α**:** The significance level of the test is 0.05.

Table 1. Parameters for the GE algorithm

Parameter	Value
Fitness function	Checkerboard
Checkerboard size	32, 72, 128, 200, 288
Fixed chromosome size	90, 210, 300, 580, 800
Initialisation	Random
Selection operation	Tournament
Tournament size	3
Replacement	Rank replacement
Max wraps	1
Generations	800
Crossover probability meta	0.9
Crossover probability solution	0.9
Mutation probability meta	0.001
Mutation probability solution	0.01

Setup. The settings in Table 1 were adopted.

The population size was determined by finding a value within 10% of where 30 runs are successful for a maximum of 800 iterations. Both chromosomes had the same initial length, roughly three times the problem size. The chromosomes were variable-length vectors of integers (2byte integers). Rank replacement is adopted with a constant population size, where the new children are pooled with the current population, ranked, and the worst individuals are removed. One-point crossover where the same crossover point is used for both parents and integer mutation where a new value was randomly chosen are used.

3.2 Results

The results for the average number of fitness evaluations are shown in Fig 3.

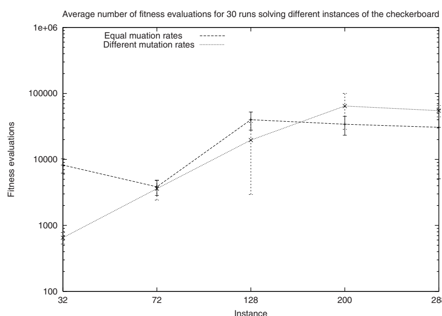


Fig. 3. On the x-axis are the problem instances, indicated by the total number of bits, and on the y-axis the number of fitness evaluations (log-scale)

After performing a right tailed t-test it was shown that there is a significant decrease in the average number of fitness evaluations required when using a lower meta chromosome mutation rate for the problem instances Cb_{32} and Cb_{128}. For these instances it is possible to reject H_0, although it is not possible to draw this conclusion for all the problems. Fig. 4 shows fitness progression over time.

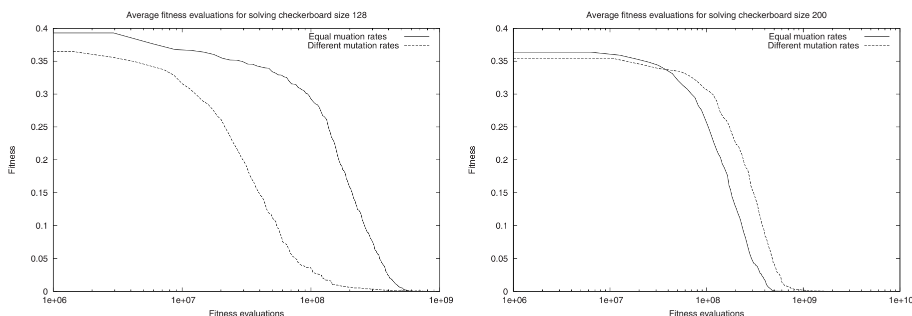


Fig. 4. Left Cb_{128}, right Cb_{200}. Log scale on x-axis and normalized y-axis. The development of best fitness during the fitness evaluations.

3.3 Experiment 2 - Sampling Each Solution Grammar n Times

In this experiment an explicit increase in sampling of the solution grammars is achieved by randomly mutating the solution chromosome, which is used to construct sentences from the evolved solution grammar. N samples of each solution grammar are generated, where N varies between 1 and 60, and the fitness of each evaluated.

Hypothesis. The performance is measured as the average number of fitness evaluations required for 30 runs to solve an instance of the Checkerboard. μ is the performance with only one sample from each chromosome. The performance of the n samples from the generated grammar is referred to as μ_{1U-nS}, where n is $2, 5, 10, 20, 60$. The false discovery rate(FDR) [11] is calculated. The p-values are derived from t-tests between μ and μ_{1U-nS}. For each instance the following hypothesis is stated:

H_0**:** None of the changes in sampling rate of the generated grammar gains significant performance to using only one sample in any of the experiments, i.e. $\mu_{1U-1S} \leq \mu_{1U-2S}$ and $\mu_{1U-1S} \leq \mu_{1U-5S}$ and $\mu_{1U-1S} \leq \mu_{1U-10S}$ and $\mu_{1U-1S} \leq \mu_{1U-20S}$ and $\mu_{1U-1S} \leq \mu_{1U-60S}$.

H_1**:** At least one of the increases in sampling of the generated grammar gains significant performance for at least one experiment, i.e. $\mu_{1U-1S} > \mu_{1U-2S}$ or $\mu_{1U-1S} > \mu_{1U-5S}$ or $\mu_{1U-1S} > \mu_{1U-10S}$ or $\mu_{1U-1S} > \mu_{1U-20S}$ or $\mu_{1U-1S} > \mu_{1U-60S}$.

α**:** The significance level of the FDR is 0.05.

Setup. The same settings are adopted as given in Table 1 with the exception that the same rate of mutation (0.01) is arbitrarily chosen and used on both the meta-grammar and solution-grammar chromosomes.

***n* individuals of each Universal Grammar.** The following algorithm is performed after a random initialization. N is the population size.

Sample. Mutate the solution chromosome with a probability of 0.01 for each individual to create n-samples. Rank the samples amongst themselves. Add the best individual to the population.
Select. Select N individuals using tournament selection.
Crossover. Crossover the N individuals to create N new individuals.
Mutate. Mutate the new individuals.
Replace. Add the new individuals to the old individuals. Rank all and remove the N worst individuals.

3.4 Results

The results for the average number of fitness evaluations are shown in Fig 5. After a right tailed t-test on the fitness evaluations to calculate p-values and then a FDR for each checkerboard size. The FDR gave only one significant test for the decrease in the number of fitness evaluations when sampling a meta grammar for the cb_{32}.

In Fig. 6 the progression of fitness during is shown. The shifting of the curves is due to the increase in the fitness evaluations required to solve larger instances.

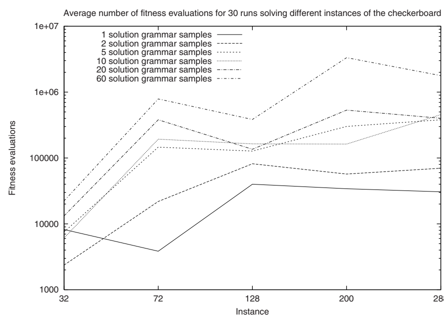


Fig. 5. On the x-axis are the problem instances, indicated by the total number of bits, and on the y-axis the number of fitness evaluations (log-scale)

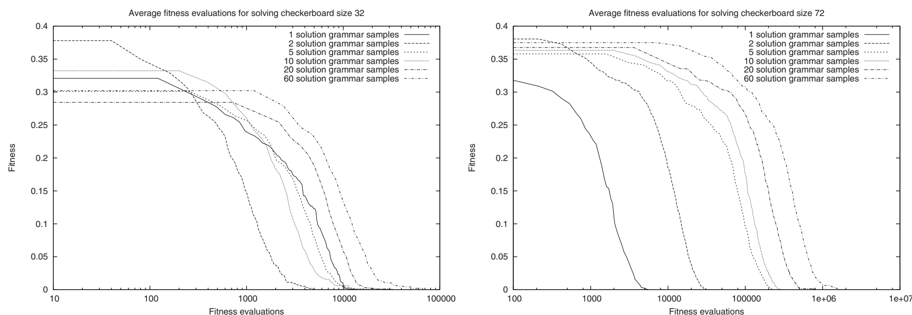


Fig. 6. Left Cb_{32}, right Cb_{72}. Log scale on x-axis. The development of best fitness during the fitness evaluations.

4 Discussion

Results for the rate of mutation experiments in Sec. 3.1 might indicate something. On three out of the five instances there is no difference in performance. It would appear that adopting a slower rate of evolution, through a lower mutation rate, on the meta-grammar chromosome can improve the performance of meta-grammar GE on the problems investigated.

The results of experiment two presented in Sec. 3.3 were initially surprising to us as we expected that an explicit increase in sampling of the evolved solution grammars would yield performance gains, but this was shown not to be the case. However if the solution grammars are considered in number of solutions they represent then there is not more search anyway. It would appear that there was no significant overall gain in performance by changing the way the algorithms explore the grammar generated by the meta grammar. What this might tell us is that most of the search is performed on the first chromosome. It has been observed in earlier studies [6,9] that the evolved solution grammars tended to

provide little choice in terms of the number of possible solutions they represented. In many cases the solutions were hard-coded into these evolved grammars, and we see similar results in this study. Below is an example of a Cb_{32} grammar that solves the problem when using equal mutation rates on the chromosomes.

```
<bitstring> ::=  <bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4>
<bbk4> ::= 1 1 0 0 | 0 0 <bit> 1
<bit> ::= 1
```

Example of a Cb_{32} grammar that solves the problem when taking 10 samples from the grammar.

```
<bitstring> ::=  <bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4> | <bbk16><bbk16>
            | <bbk2><bbk2><bbk2><bbk2><bbk2><bbk2><bbk2><bbk2>
              <bbk2><bbk2><bbk2><bbk2><bbk2><bbk2><bbk2><bbk2>
<bbk16> ::= 1 0 1 1 1 0 <bit> 1 <bit> 1 <bit> <bit> 0 0 1 1
        | 1 <bit> 0 0 <bit> 1 0 <bit> 0 <bit> 0 <bit> <bit> <bit> <bit> 1
<bbk4> ::= <bit> 1 0 0  |  0 0 <bit> 1  |  1 0 <bit> 0
<bbk2> ::= <bit> 0  |  0 <bit>  |  0 <bit>
<bit> ::= 1
```

These results show that the blocks of useful code evolved into the solution grammar sometimes match the pattern completely. Given this evidence and the results of the experiments conducted in this study, a more detailed analysis of the the meta-grammar approach was required. In particular we wish to better understand the sizes of the evolved solution grammars and the relative amount of search being undertaken on the meta-grammar and solution grammar chromosomes.

4.1 Individual Inspection

The individual solutions generated were scrutinized in order to further investigate the suggestion that most search is performed on the first chromosome.

Codons Used. The average length and fraction of codons used at the end of each run are shown in Fig. 7. The number of codons used in the solution grammar chromosome does not increase as much as the meta grammar chromosome when the problems sizes increase, and there are a lot more expressed codons on the meta-grammar chromosome than on the solution grammar chromosome. When we compare the chromosomes lengths and expressed lengths for the mutation rate experiment no significant difference is observed. This analysis would suggest performance could be increased by using a different initialization operation that balanced the use of the two chromosomes.

Possible Output Strings from the Solution Grammar. The solutions grammars seem to have quite few paths through them. A path describes a sequence of productions which creates an output string. Combining the paths with the number of codons used gives the set of non-unique possible output strings. These are shown in Fig. 8 and 9. It can be seen that the possible output strings of the grammars are not infinite and sometimes as low as 4.

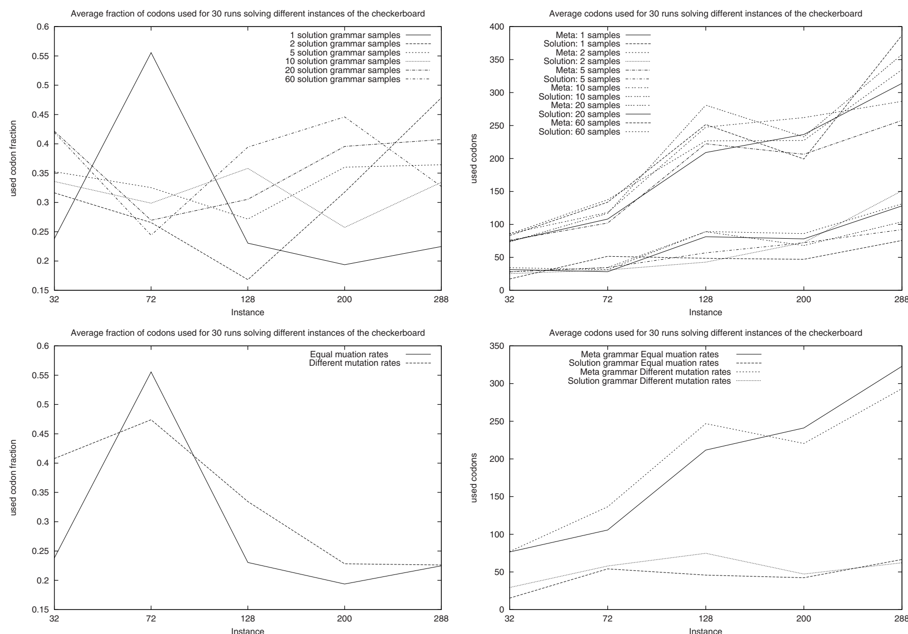


Fig. 7. Left is the average fraction of codon use at the end point of each instance. Right is the average length of each codon used at the end of each instance. Top is for n-samples and bottom is for different mutation rates.

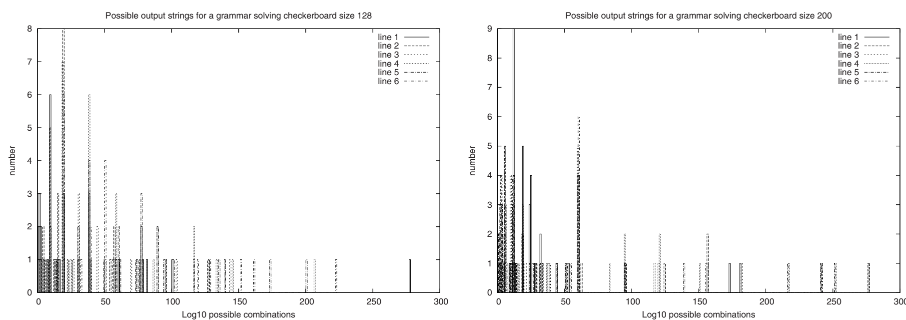


Fig. 8. Left Cb_{128}, right Cb_{200}. Histogram over the number of combinations of the possible paths in the solutions grammar and codons used. Samples with "Infinite" value are put in the rightmost bin. Line 1 is 1U-1S, Line 2 is 1U-2S, Line 3 is 1U-5S, Line 4 is 1U-10S, Line 5 is 1U-20S, Line 6 is 1U-60S.

It is therefore clear that the evolved solution grammars do not represent a large proportion of the overall search space as covered by the original meta-grammar, and the number of solutions represented tends to be very small. Despite this the

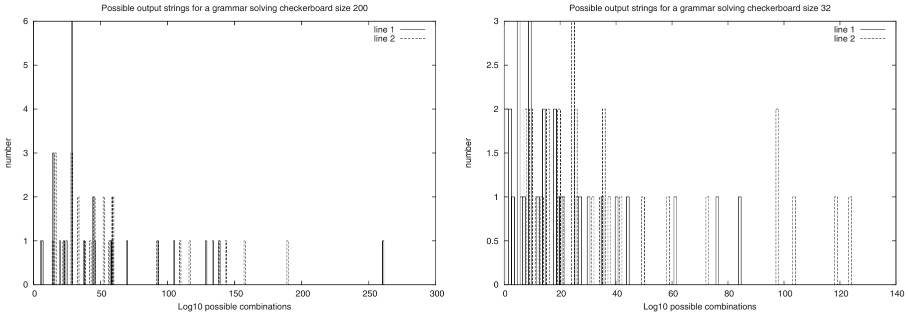


Fig. 9. Left Cb_{288}, right Cb_{32}. Histogram over the number of combinations of the possible paths in the solutions grammar and codons used. Samples with "Infinite" value are put in the rightmost bin. Line 1 is 0.01mU-0.01mS, Line 2 is 0.001mU-0.01mS.

performance of meta-grammar GE is impressive, and this analysis provides us a guide future research to improve the performance of meta-grammar GE.

5 Conclusion and Future Work

An analysis of altering the rate of sampling of the evolved solution grammars in meta grammar GE is undertaken. Two approaches were examined, the first adopts implicit sampling using different rates of mutation on the evolved solution grammar versus the solutions sampled from the evolved solution grammar. The second approach explicitly generates more than one sample from each solution grammar in a kind of local-search by randomly mutating the solution chromosome, which is used to construct sentences from the evolved solution grammar. On the problem instances examined neither approach was found to conclusively improve the performance of the meta-grammar approach to GE in terms of the number of fitness evaluations to find a solution. It is found that the majority of the evolutionary search is currently focused on the generation of the solution grammars to such an extent that the candidate solutions are often hard-coded into them making the solution chromosome effectively redundant. This opens the door to future work in which we can explore how the search can be better balanced between the meta and solution grammars, and the possibility of further performance gains. Further experimentation is needed to see if the same effects occur on other problem types e.g. Trap functions, dynamic functions. A study of the initialisation will be undertaken, and of measures adopted to understand the form of the evolved blocks of code in the evolving solutions grammars.

Acknowledgement

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

References

1. O'Neill, M., Ryan, C.: Grammatical Evolution by Grammatical Evolution: The Evolution of Grammar and Genetic Code. In: Keijzer, M., O'Reilly, U.-M., Lucas, S.M., Costa, E., Soule, T. (eds.) EuroGP 2004. LNCS, vol. 3003, pp. 138–149. Springer, Heidelberg (2004)
2. O'Neill, M., Ryan, C.: Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Kluwer, Dordrecht (2003)
3. Brabazon, A., O'Neill, M.: Biologically Inspired Algorithms for financial Modelling. Springer, Heidelberg (2006)
4. Cleary, R., O'Neill, M.: An attribute grammar decoder for the 01 multiconstrained knapsack problem. In: Raidl, G.R., Gottlieb, J. (eds.) EvoCOP 2005. LNCS, vol. 3448, pp. 34–45. Springer, Heidelberg (2005)
5. Shan, Y., McKay, R.I., Baxter, R., Abbass, H., Essam, D., Hoai, N.X.: Grammar model-based program evolution. In: Proc. of CEC 2004, pp. 478–485. IEEE Press, Los Alamitos (2004)
6. O'Neill, M., Brabazon, A.: mGGA: The meta-grammar genetic algorithm. In: Keijzer, M., Tettamanzi, A.G.B., Collet, P., van Hemert, J.I., Tomassini, M. (eds.) EuroGP 2005. LNCS, vol. 3447, pp. 311–320. Springer, Heidelberg (2005)
7. Dempsey, I., O'Neill, M., Brabazon, A.: Meta-grammar constant creation with grammatical evolution by grammatical evolution. In: Beyer, H.-G., O'Reilly, U.-M. (eds.) Proc. of GECCO 2005, vol. 2, pp. 1665–1671. ACM Press, New York (2005)
8. Hemberg, E., Gilligan, C., O'Neill, M., Brabazon, A.: A grammatical genetic programming approach to modularity in genetic algorithms. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 1–11. Springer, Heidelberg (2007)
9. Dempsey, I.: Grammatical Evolution in Dynamic Environments. PhD thesis, University College Dublin (2007)
10. Garibay, O.O., Garibay, I.I., Wu, A.S.: The modular genetic algorithm: Exploiting regularities in the problem space. In: Yazıcı, A., Şener, C. (eds.) ISCIS 2003. LNCS, vol. 2869, pp. 584–591. Springer, Heidelberg (2003)
11. Benjamini, Y., Hochberg, Y.: Controlling the false discovery rate: a practical and powerful approach to multiple testing. Journal of the Royal Statistical Society B 57(1), 289–300 (1995)

Author Index

Badran, Khaled 301
Banzhaf, Wolfgang 73, 182
Brabazon, Anthony 362

Dignum, Stephen 110, 158, 312
Doucette, John 266
Downing, Richard M. 194

Ebner, Marc 61

Flanagan, Colin 37
Fonlupt, Cyril 98

Galván-López, Edgar 312
Geihs, Kurt 254
Girgin, Sertan 218
Graff, Mario 122

Heckendorn, Robert B. 1
Hemberg, Erik 362
Heywood, Malcolm I. 266, 289
Hutchison, Tyler 134

Jackson, David 86, 170
Jaśkowski, Wojciech 13

Krawiec, Krzysztof 13

Langdon, W.B. 73
Larkin, Fiacc 49
Li, HongYu 325

Marion-Poty, Virginie 98
McIntyre, Andrew R. 289
McPhee, Nicholas Freitag 134, 206
Miconi, Thomas 25
Montes de Oca, Marco A. 278
Murphy, Gearoid 146

Naidoo, Amashini 350
Neshatian, Kourosh 242
Niranjan, Mahesan 325

O'Neill, Michael 362
Ohs, Brian 134

Pillay, Nelishia 350
Poli, Riccardo 110, 122, 158, 206, 312
Preux, Philippe 218

Raja, Adil 37
Robilliard, Denis 98
Rockett, Peter 301, 325
Ryan, Conor 49, 146

Sekanina, Lukas 230
Soule, Terence 1

Thomason, Russell 1

Vasicek, Zdenek 230

Weise, Thomas 254
Wieloch, Bartosz 13
Wilson, Garnett 182

Yamamoto, Lidia 337

Zapf, Michael 254
Zhang, Mengjie 242
Zhang, Yang 325