

Genetic and Evolutionary Computation

Stephan M. Winkler

Wolfgang Banzhaf

Ting Hu

Alexander Lalejini *Editors*

Genetic Programming Theory and Practice XXI

 Springer

Genetic and Evolutionary Computation

Series Editors

Wolfgang Banzhaf , Department of Computer Science and Engineering, Michigan State University, East Lansing, USA

Kalyanmoy Deb , Department of Electrical and Computer Engineering, Michigan State University, East Lansing, USA

Mengjie Zhang, School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand

The area of Genetic and Evolutionary Computation has seen an explosion of interest in recent years. Methods based on the variation-selection loop of Darwinian natural evolution have been successfully applied to a whole range of research areas.

The Genetic and Evolutionary Computation Book Series publishes research monographs, edited collections, and graduate-level texts in one of the most exciting areas of Computer Science. As researchers and practitioners alike turn increasingly to search, optimization, and machine-learning methods based on mimicking natural evolution to solve problems across the spectrum of the human endeavor, this growing field will continue to surprise with novel applications and results. Recent award-winning PhD theses, special topics books, workshops and conference proceedings in the areas of EC and Artificial Life Studies are of interest.

Areas of coverage include applications, theoretical foundations, technique extensions and implementation issues of all areas of genetic and evolutionary computation. Topics may include, but are not limited to:

Optimization (multi-objective, multi-level) Design, control, classification, and system identification Data mining and data analytics Pattern recognition and deep learning Evolution in machine learning Evolvable systems of all types Automatic programming and genetic improvement

Proposals in related fields such as:

Artificial life, artificial chemistries Adaptive behavior and evolutionary robotics Artificial immune systems Agent-based systems Deep neural networks Quantum computing will be considered for publication in this series as long as GEVO techniques are part of or inspiration for the system being described. Manuscripts describing GEVO applications in all areas of engineering, commerce, the sciences, the arts and the humanities are encouraged.

Prospective Authors or Editors:

If you have an idea for a book, we would welcome the opportunity to review your proposal. Should you wish to discuss any potential project further or receive specific information regarding our book proposal requirements, please contact Wolfgang Banzhaf, Kalyan Deb or Mio Sugino:

Areas: Genetic Programming/other Evolutionary Computation Methods, Machine Learning, Artificial Life

Wolfgang Banzhaf Consulting Editor BEACON Center for Evolution in Action Michigan State University, East Lansing, MI 48824 USA banzhafw@msu.edu

Areas: Genetic Algorithms, Optimization, Meta-Heuristics, Engineering

Kalyanmoy Deb Consulting Editor BEACON Center for Evolution in Action Michigan State University, East Lansing, MI 48824 USA kdeb@msu.edu

Mio Sugino mio.sugino@springer.com

The GEVO book series is the result of a merger the two former book series: Genetic Algorithms and Evolutionary Computation <https://www.springer.com/series/6008> and Genetic Programming <https://www.springer.com/series/6016>.

Stephan M. Winkler · Wolfgang Banzhaf ·
Ting Hu · Alexander Lalejini
Editors

Genetic Programming Theory and Practice XXI

Editors

Stephan M. Winkler 
School of Informatics, Communications
and Media
University of Applied Sciences Upper
Austria
Hagenberg, Austria

Ting Hu 
School of Computing
Queen's University
Kingston, ON, Canada

Wolfgang Banzhaf 
Computer Science and Engineering
Michigan State University
East Lansing, MI, USA

Alexander Lalejini
School of Computing
Grand Valley State University
Allendale, MI, USA

ISSN 1932-0167 ISSN 1932-0175 (electronic)
Genetic and Evolutionary Computation
ISBN 978-981-96-0076-2 ISBN 978-981-96-0077-9 (eBook)
<https://doi.org/10.1007/978-981-96-0077-9>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2025

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd. The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

If disposing of this product, please recycle the paper.

Preface

In 2022, the first post-COVID edition of the Genetic Programming Theory and Practice (GPTP) workshop was held at University of Michigan; following the now established tradition of alternating between UM and MSU, we returned to Michigan State University in 2023. In 2024, GPTP XXI was held back at University of Michigan in June 2024; approximately 50 GPTP'ers gathered at Weiser Hall, making GPTP 2024 a big success.

As usual, we started on Thursday with a great keynote by Allison Earl (University of Michigan, Psychology) who talked about the principles of persuasion and what changes minds, followed by vivid discussions about how human psychology influences decision making and how we might more effectively advocate for GP approaches to a broader audience. Over the day, we then saw and discussed talks given by Mark Kotanchek (Evolved Analytics) and Mengjie Zhang (Victoria University of Wellington) as well as lightning talks by Erik Hemberg (MIT Computer Science and Artificial Intelligence Lab), Stephen Kelly (McMaster University), Moshe Sipper (Ben-Gurion University of the Negev), Nuno Lourenço (University of Coimbra), Edward Pantridge (Spindle Health), Illya Bakurov (Michigan State University), and Oscar Garnica (Universidad Complutense de Madrid). Additionally, we had an intense discussion session about universal GP implementations, standards for GP components, and GP and deep learning.

The second day started with an exciting talk by María Rebolleda-Gómez (University of California, Irvine) on engineering the environment to shape microbial community functions. María highlighted areas where ideas and techniques honed in the GP community might inspire useful new methods in enhancing microbial community functions in the laboratory. The keynote was followed by talks by Lukas Sekanina (Brno University of Technology), Lee Spector (Amherst College), and Leonardo Vanneschi (Universidade Nova de Lisboa). In the afternoon, lightning talks were presented by Bogdan Burlacu (University of Applied Sciences Upper Austria, HEAL), Chris Crary (University of Florida), Yuri Lavinias (University of Toulouse), Erik Fredericks (Grand Valley State University), and Anselmo Pontes (Autogenetics Research Lab). We had a lively discussion about generative art, and what it means to claim something as art.

On Saturday, we concluded GPTP 2024 with a fascinating keynote given by Kamaludin Dingle (Gulf University for Science & Technology) on simplicity bias, algorithmic complexity, and biological evolution and talks by Jose Guadalupe Hernandez (Cedars-Sinai) and Charles Ofria (Michigan State University). Once again, we all have to be grateful for the great talks we saw and also the great discussions we had throughout the event. The hype machine learning is seeing at the moment is continuing. It is also a great opportunity for promoting GP as a well-suited method for many applications, especially when interpretable results are sought for.

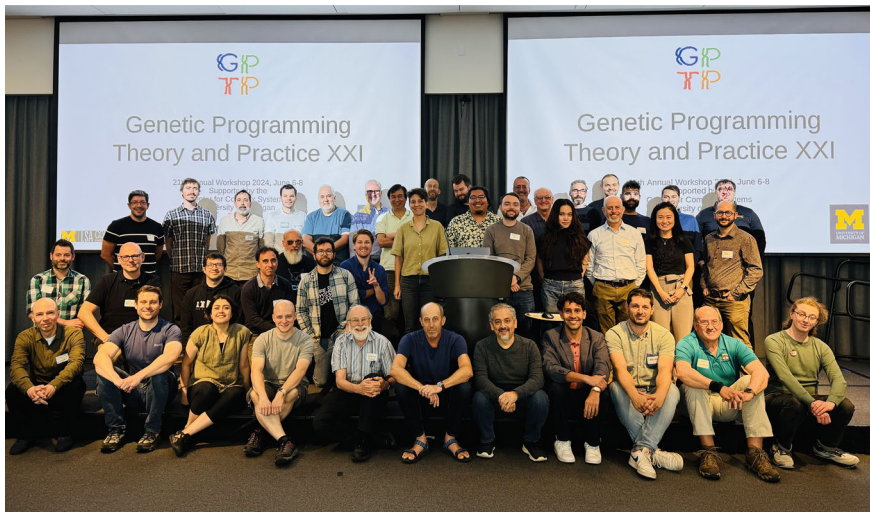


Fig. 1 Attendees of GPTP XXI at University of Michigan, June 2024

We are convinced that GPTP will continue to be a core event for genetic programming research, bringing together academics, practitioners and theorists from diverse fields of science that intersect in our community, providing for a constructive, thoughtful, inspired and open interchange of ideas.

East Lansing, MI, USA
Kingston, ON, Canada
Allendale, MI, USA
Hagenberg, Austria
September 2024

Wolfgang Banzhaf
Ting Hu
Alexander Lalejjini
Stephan M. Winkler

Acknowledgements

We would like to thank all of the participants for making our “GP—Theory and Practice” workshop a successful in-person event once again in 2024. Special thanks go to our three wonderful keynote speakers, Allison, Maria, and Kamaludin for their inspiring talks! Special thanks to Fabricio Olivetti de Franca who kept us caffeinated by brewing fantastic coffee with fresh coffee beans from Brazil.

We would also like to thank our financial supporters for making the existence of GP Theory and Practice possible for twenty-one great editions. For 2024, we are grateful to the following sponsors:

- Michael Affenzeller from HEAL and the University of Applied Sciences Upper Austria
- Stuart Card
- John Koza
- Jason H. Moore at the Department of Computational Biomedicine in Cedars-Sinai.

A number of people made key contributions to the organization of the workshop. We thank the Center for the Study of Complex Systems at the University of Michigan for hosting GPTP once again. With a heavy heart we especially thank Linda Wood, who helped behind the scenes before, during and after the workshop—for the last time, as she has retired in the meanwhile.

We are particularly grateful for contractual assistance by Mio Sugino, Springer-Nature Tokyo, and editorial assistance by Springer-Nature Chennai.

East Lansing, MI, USA
Kingston, ON, Canada
Allendale, MI, USA
Hagenberg, Austria
September 2024

Wolfgang Banzhaf
Ting Hu
Alexander Lalejini
Stephan M. Winkler

Contents

1	Representation and Reachability: Assumption Impact in Data Modeling	1
	Mark Kotanchek, Nathan Haut, and Kelvin Kotanchek	
2	EvoFeat: Genetic Programming-Based Feature Engineering Approach to Tabular Data Classification	27
	Hengzhe Zhang, Qi Chen, Bing Xue, Yan Wang, Aimin Zhou, and Mengjie Zhang	
3	Deep Learning-Based Operators for Evolutionary Algorithms	51
	Eliad Shem-Tov, Moshe Sipper, and Achiya Elyasaf	
4	Survey of Genetic Programming and Large Language Models	67
	Erik Hemberg, Steven Jorgensen, and Una-May O'Reilly	
5	Evolving Many-Model Agents with Vector and Matrix Operations in Tangled Program Graphs	87
	Tanya Djavaheerpour, Ali Naqvi, Eddie Zhuang, and Stephen Kelly	
6	Automatic Design of Autoencoders Using NeuroEvolution	107
	Alexy de Almeida, Nuno Lourenço, and Penousal Machado	
7	Code Building Genetic Programming is Faster than PushGP	133
	Edward Pantridge and Thomas Helmuth	
8	Sharpness-Aware Minimization in Genetic Programming	151
	Ilyia Bakurov, Nathan Haut, and Wolfgang Banzhaf	
9	Tree-Based Grammatical Evolution with Non-Encoding Nodes	177
	Marina de la Cruz, Oscar Garnica, J. Manuel Velasco, Daniel Parra, and J. Ignacio Hidalgo	
10	Genetic Programming with Memory for Approximate Data Reconstruction	199
	Lukas Sekanina and Tadeas Juza	

11	Ratcheted Random Search for Self-programming Boolean Networks	219
	Lee Spector	
12	Exploring Non-bloating Geometric Semantic Genetic Programming	237
	Leonardo Vanneschi, Davide Farinati, Diogo Rasteiro, Liah Rosenfeld, Gloria Pietropolli, and Sara Silva	
13	Revisiting Gradient-Based Local Search in Symbolic Regression	259
	Bogdan Burlacu, Stephan M. Winkler, and Michael Affenzeller	
14	It's Time to Revisit the Use of FPGAs for Genetic Programming	275
	Christopher Crary, Greg Stitt, Bogdan Burlacu, and Wolfgang Banzhaf	
15	Interpretable Genetic Programming Models for Real-World Biomedical Images	297
	Yuri Lavinas, Camilo De La Torre, K��vin Cortacero, Dennis G. Wilson, and Sylvain Cussat-Blanc	
16	Crafting Generative Art Through Genetic Improvement: Managing Creative Outputs in Diverse Fitness Landscapes	321
	Erik M. Fredericks, Denton Bobeldyk, and Jared M. Moore	
17	Cell Regulation and the Early Evolution of Autonomous Control	337
	Anselmo C. Pontes and Nicholas L. Panchy	
18	How to Measure Explainability and Interpretability of Machine Learning Results	357
	Elisabeth Mayrhuber, Bogdan Burlacu, and Stephan M. Winkler	
19	Lexicase Selection Parameter Analysis: Varying Population Size and Test Case Redundancy with Diagnostic Metrics	375
	Jose Guadalupe Hernandez, Anil Kumar Saini, and Jason H. Moore	
20	Using Lineage Age to Augment Search Space Exploration in Lexicase Selection	395
	Karen Suzue, Charles Ofria, and Alexander Lalejini	
	Index	413

Contributors

Michael Affenzeller Department of Software Engineering, University of Applied Sciences Upper Austria, Hagenberg, Austria

Ilya Bakurov Department of Computer Science and Engineering, Michigan State University, East Lansing, MI, USA

Wolfgang Banzhaf Department of Computer Science and Engineering, Michigan State University, East Lansing, MI, USA

Denton Bobeldyk College of Computing, Grand Valley State University, Allendale, MI, USA

Bogdan Burlacu Department of Software Engineering, University of Applied Sciences Upper Austria, Hagenberg, Austria;
Heuristic and Evolutionary Algorithms Laboratory, University of Applied Sciences Upper Austria, Hagenberg, Upper Austria, Austria

Qi Chen The Centre for Data Science and Artificial Intelligence & School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand

Kévin Cortacero CRCT - INSERM, Toulouse, France

Christopher Crary Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA

Sylvain Cussat-Blanc IRIT - CNRS UMR5505, University Toulouse Capitole, Institut Universitaire de France, Toulouse, France

Alexy de Almeida CISUC/LASI, DEI, University of Coimbra, Coimbra, Portugal

Marina de la Cruz Department of Computer Architecture, Universidad Complutense de Madrid, Madrid, Spain

Tanya Djavaherpour Department of Computing and Software, McMaster University, Hamilton, Canada

Achiya Elyasaf Department of Software and Information Systems Engineering,
Ben-Gurion University of the Negev, Beer-Sheva, Israel

Davide Farinati NOVA Information Management School (NOVA IMS),
Universidade Nova de Lisboa, Lisboa, Portugal

Erik M. Fredericks College of Computing, Grand Valley State University,
Allendale, MI, USA

Oscar Garnica Department of Computer Architecture, Universidad Complutense
de Madrid, Madrid, Spain

Nathan Haut Department of Computational Mathematics, Science and
Engineering, Michigan State University, East Lansing, MI, USA;
Department of Computer Science and Engineering, Michigan State University,
East Lansing, MI, USA

Thomas Helmuth Computer Science, Hamilton College, Clinton, NY, USA

Erik Hemberg ALFA, MIT CSAIL, Cambridge, MA, USA

Jose Guadalupe Hernandez Cedars-Sinai Medical Center, Los Angeles, CA,
USA

J. Ignacio Hidalgo Department of Computer Architecture, Universidad
Complutense de Madrid, Madrid, Spain

Steven Jorgensen ALFA, MIT CSAIL, Cambridge, MA, USA

Tadeas Juza Faculty of Information Technology, Brno University of Technology,
Brno, Czech Republic

Stephen Kelly Department of Computing and Software, McMaster University,
Hamilton, Canada

Kelvin Kotanchek Evolved Analytics LLC, Rancho Santa Fe, CA, USA

Mark Kotanchek Evolved Analytics LLC, Rancho Santa Fe, CA, USA

Camilo De La Torre IRIT - CNRS UMR5505, University Toulouse Capitole,
Toulouse, France

Alexander Lalejini School of Computing, Grand Valley State University,
Allendale, MI, USA

Yuri Lavinias IRIT - CNRS UMR5505, University Toulouse Capitole, Toulouse,
France

Nuno Lourenço CISUC/LASI, DEI, University of Coimbra, Coimbra, Portugal

Penousal Machado CISUC/LASI, DEI, University of Coimbra, Coimbra,
Portugal

Elisabeth Mayrhuber Bioinformatics Research Group, PLFDoc, University of Applied Sciences Upper Austria, Hagenberg, Austria;
Computer Vision Lab, TU Wien, Vienna, Austria

Jared M. Moore College of Computing, Grand Valley State University,
Allendale, MI, USA

Jason H. Moore Cedars-Sinai Medical Center, Los Angeles, CA, USA

Ali Naqvi Department of Computing and Software, McMaster University,
Hamilton, Canada

Charles Ofria Department of Computer Science and Engineering and Program in Ecology, Evolution and Behavior, Michigan State University, East Lansing, MI, USA

Una-May O'Reilly ALFA, MIT CSAIL, Cambridge, MA, USA

Nicholas L. Panchy Institute for Cyber Enabled Research, Michigan State University, East Lansing, USA

Edward Pantridge Spindle Health, Boston, MA, USA

Daniel Parra Department of Computer Architecture, Universidad Complutense de Madrid, Madrid, Spain

Gloria Pietropoli Department of Mathematics, Informatics, and Geosciences, University of Trieste, Trieste, Italy

Anselmo C. Pontes Autogenetics Research Lab, Washington DC, USA

Diogo Rasteiro NOVA Information Management School (NOVA IMS), Universidade Nova de Lisboa, Lisboa, Portugal

Liah Rosenfeld NOVA Information Management School (NOVA IMS), Universidade Nova de Lisboa, Lisboa, Portugal

Anil Kumar Saini Cedars-Sinai Medical Center, Los Angeles, CA, USA

Lukas Sekanina Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic

Eliad Shem-Tov Department of Software and Information Systems Engineering, Ben-Gurion University of the Negev, Beer-Sheva, Israel

Sara Silva LASIGE, Department of Informatics, Faculty of Sciences, University of Lisbon, Lisbon, Portugal

Moshe Sipper Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel

Lee Spector Amherst College, Amherst, MA, USA

Greg Stitt Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA

Karen Suzue Department of Computer Science and Engineering and Program in Ecology, Evolution and Behavior, Michigan State University, East Lansing, MI, USA

Leonardo Vanneschi NOVA Information Management School (NOVA IMS), Universidade Nova de Lisboa, Lisboa, Portugal

J. Manuel Velasco Department of Computer Architecture, Universidad Complutense de Madrid, Madrid, Spain

Yan Wang School of Computing, Macquarie University, Sydney, NSW, Australia

Dennis G. Wilson ISAE-SUPAERO, Toulouse, France

Stephan M. Winkler Department of Medical and Bioinformatics, University of Applied Sciences Upper Austria, Hagenberg, Austria

Bing Xue The Centre for Data Science and Artificial Intelligence & School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand

Hengzhe Zhang The Centre for Data Science and Artificial Intelligence & School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand

Mengjie Zhang The Centre for Data Science and Artificial Intelligence & School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand

Aimin Zhou Shanghai Institute of AI for Education, School of Computer Science and Technology, East China Normal University, Shanghai, China

Eddie Zhuang Department of Computing and Software, McMaster University, Hamilton, Canada

Chapter 1

Representation and Reachability: Assumption Impact in Data Modeling



Mark Kotanchek, Nathan Haut, and Kelvin Kotanchek

Abstract Data modeling implicitly makes assumptions. If those assumptions match the true system dynamics, the resulting model will be both insightful and predictive. If assumption alignment is not achieved, then we induce risk in using those models for prediction and action. Herein we argue that a good model consolidates energy (i.e., is parsimonious) and properly spans the interstitial regions between observed behavior points. From that viewpoint, we review and assess some of the common assumptions made in data modeling, in general, as well as in symbolic regression, in particular. We consider aspects of risk reduction, quality measures, functional building blocks, model representation, search operators, selection strategies, and trustability.

1.1 Introduction

The first principle is that you must not fool yourself—and you are the easiest person to fool.
—Richard Feynman (1918–1988), Caltech commencement address, 1974

Every modeling approach makes assumptions about the system being modeled. Sometimes those assumptions are correct. Almost all of the time, however, the assumptions are wrong to some degree. However, our goal as practitioners is to be good-enough in our assumptions.

Where we get into trouble is when we do not check our assumption validity and, instead, reflexively use tools because they are familiar, have popular mind-share or mesh well with domain dogma.

M. Kotanchek · K. Kotanchek
Evolved Analytics LLC, Rancho Santa Fe, CA, USA
e-mail: mark@evolved-analytics.com

K. Kotanchek
e-mail: kelvin@evolved-analytics.com

N. Haut (✉)
Department of Computational Mathematics, Science and Engineering, Michigan State University,
East Lansing, MI, USA
e-mail: hautnath@msu.edu

1.1.1 Energy Consolidation and Parsimony

As illustrated in Fig. 1.1, a good model essentially consolidates the energy of the observations into a more parsimonious form which gives us more insight than simply examining the raw data. The ability of a model to consolidate energy is dependent upon the mechanisms we provide to the modeling process

1.1.2 Interstitial Spaces

There are an infinite number of models that will exactly fit any finite data set. There is an even more infinite set of models which will approximately fit a data set. Figure 1.2 illustrates this situation with a single variable and two observed data points. The key out of that graphic is that the purpose of a model is to fill in the interstitial gaps between the observed data points. Adding dimensions (variables) geometrically increases the number of interstitial gaps that must be spanned.

Modeling assumptions define the possibilities and constraints of the behavior in the interstitial gaps as well as in extrapolation.

1.1.3 Model Representation

If all you have is a hammer, everything looks like a nail.
—Abraham Maslow, *The Psychology of Science*, 1966

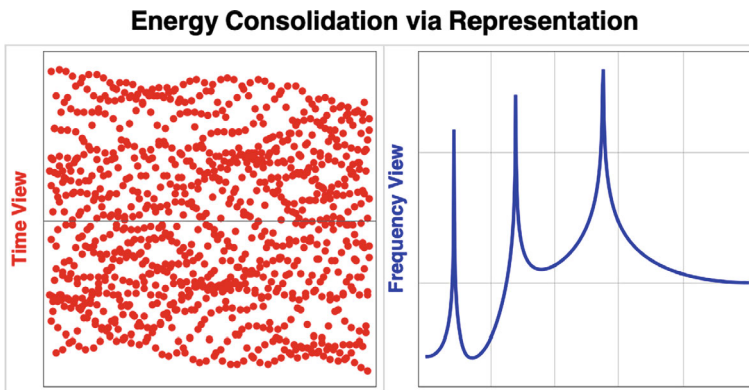


Fig. 1.1 Applying a Fourier transform to time series data can consolidate the energy which, in turn, provides insight into the underlying system. Energy consolidation is the foundation of much of signal processing

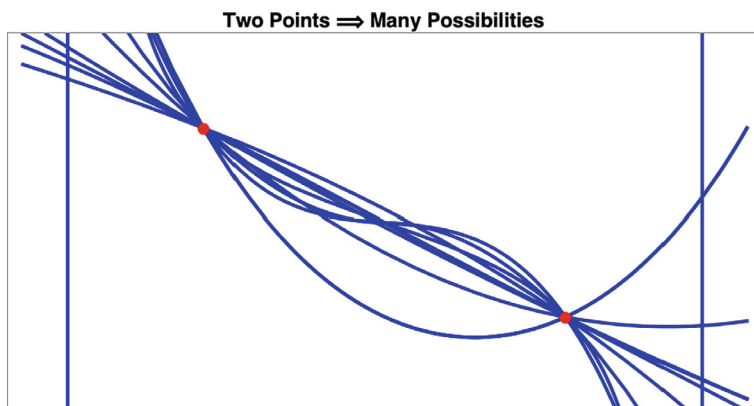


Fig. 1.2 All of these models exactly fit the two observed data points. Choosing between the possibilities requires assumptions about proper behavior between the observables

Figure 1.3 shows a taxonomy of common modeling techniques partitioned by the foundation of their modeling representation. The data record approach uses observations either by taking the closest point or by placing kernels (e.g., Gaussians) on selected data points. Data splits assume that pairwise decompositions of variables will allow accurate prediction. Network models assume a flow through transfer functions—typically sigmoids although GMDH (Group Methods for Data Handling) uses polynomials. Algebraic modeling tries to develop global models with explicit variables.

Most of the popular modeling techniques do well when operating near observed data points (otherwise, they would not be popular). Figure 1.4 illustrates some of their assumptions and limitations. Assumptions of linearity cause issues if the targeted system is not linear. Fortunately for much of engineering and science, the linear approximation will be valid if the operating range is sufficiently constrained. Neural networks and related techniques are universal approximators but have the disadvantage that they are black boxes and teasing out variable importance is difficult. Random Forests and their cousins like Gradient Boosted Trees create ensembles of weak learners which, in aggregate, often do a remarkably good job of prediction. They tend to struggle if spurious variables are present and are black boxes which makes variable prioritization and insight difficult. Gaussian Processes, like Support Vector Machines, place kernels on observed data points and fill in the interstitial gaps by relying upon the exponential decay of their kernels. Mixed kernels of gaussians and linear components help to provide better extrapolation capabilities. Although still black boxes, they are attractive in that they do have a trust metric associated with their predictions. Symbolic regression assumes explicit algebraic models with a few explicit variables will capture the system behavior; however, we have to provide the modeling with the proper operators and time to let evolutionary search discover model forms. This discussion is summarized in the table of Fig. 1.5.

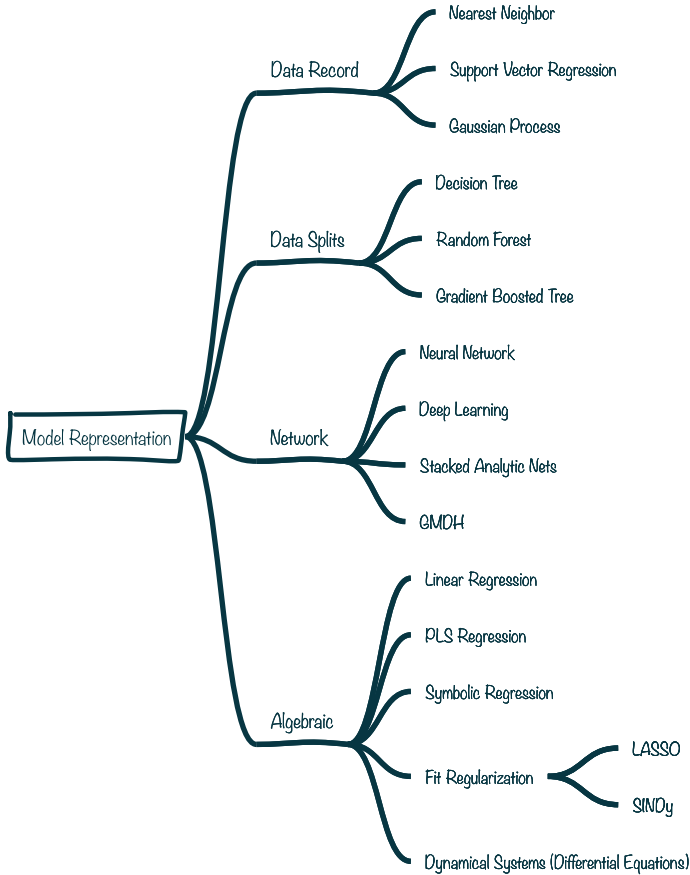


Fig. 1.3 Data modeling techniques can be broadly grouped by how they incorporate data into their resulting models. Each approach has strengths and weaknesses as is illustrated in Fig. 1.4

1.2 Algebraic Modeling

The creator of the universe works in mysterious ways. But he uses a base ten counting system and likes round numbers.

—Scott Adams

From a practitioner standpoint, algebraic models are very attractive due to both their transparency and their ease of deployment for prediction and soft sensors. There are three linear techniques which will produce algebraic models as the nonlinear symbolic regression which are discussed in the following subsections. The taxonomy we will use is shown in Fig. 1.6.

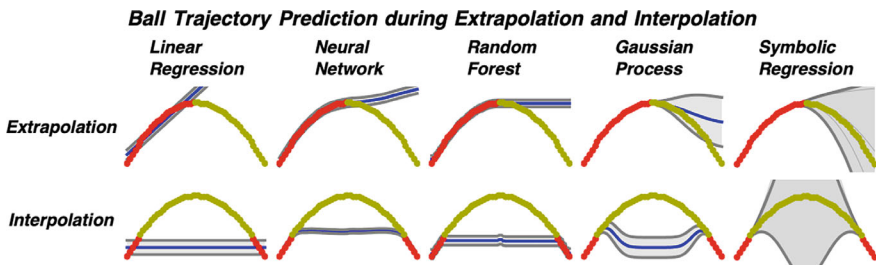


Fig. 1.4 The Nate Ball problem illustrates model assumptions from available (red) observations and the prediction of the unobserved (yellow) behavior. Although symbolic regression predicts well, the key attribute is that the ensemble knows when it is extrapolating and caution should be applied in using the prediction. Often when data is sparse, we are extrapolating even when nominally interpolating in terms of the data ranges

Modeling Technique Pros & Cons		
Technique	Pros	Cons
Linear	Transparent models Computationally Simple Mathematical Foundations	Linearity Assumption Struggles with Correlated Variables Variable Selection is Difficult
Neural Networks	Universal Approximator	Black Box Struggles with Extrapolation Requires Lots of Data
Random Forest Gradient Boosted Trees	Limited Assumptions	Extrapolation Problems Black Box Assumes all variables matter
Gaussian Process	Detects Extrapolation	Black Box Assumes All Variables Matter
Symbolic Regression	Transparent Models Automatic Variable Selection Variable Options Trustable Models	Assumes relatively few variables matter Computationally Intensive Assumes Algebraic Model

Fig. 1.5 This summarizes the Modeling Technique Pros & Cons

1.2.1 Linear Modeling

Using a term like nonlinear science is like referring to the bulk of zoology as the study of non-elephant animals.
—Stan Ulam (attributed)

The expression $f = a_1x_1 + a_2x_2 + a_3x_3 + \dots$ is truly linear with the coefficients normally found using a least-squares fit. Although this hyperplane model is attractive in the sense that it is guaranteed to be bounded when evaluated, the least-squares fit has several issues since it implicitly assumes that all of the inputs should be used. This causes issues if we have ill-conditioned data such as is often encountered with biological systems such as metabolic data where we might have thousands of available variables and fewer than hundreds of observations (aka, patients). Such situations are also populated with correlated variables which also cause issues.



Fig. 1.6 Underneath the algebraic modeling umbrella, we have three variants founded on linearity assumptions along with symbolic regression which can relax those assumptions

That said, the statistical rules are that any expression of the form $f(x) = f_1(x) + f_2(x) + \dots$ is also considered to be a linear expression so we can get a nonlinear response from a linear expression. The $f_i(x)$ can be viewed as linearizing transforms essentially mapping the input variables into a space where standard statistical tools can be deployed [18]. Often, these transforms are polynomials; however, any basis set of input transforms can be used.

From a classically-trained perspective, this means that developed models can be validated using mathematical tools having hundreds of years of development, e.g., ANOVA Tables and such. Unfortunately, identifying the proper linearizing transforms is nontrivial. Two mechanisms are discussed later: fit regularization and symbolic regression.

1.2.2 *PLS Regression*

Principle Components Analysis (PCA) and Singular Value Decompositions (SVD) will partition a data array into its dominant linear representation—effectively creating a new orthogonal coordinate system that captures most of the observed input and response behavior. Although this is nice for data clustering, it does not address the desired objective of identifying the input variables that create the observed response. Fortunately, Wold developed Partial Least Squares Regression, PLSR—subsequently renamed to be a more accurate Projection to Latent Spaces—decomposition which does a decomposition on the input variables based upon their contribution to the observed response [20].

This is a linear representation; however, it is quite powerful since it can identify the most important *linear* variables driving the observed behavior and is the only linear technique which can handle the aforementioned biological data sets. Furthermore, the focusing can be applied iteratively as illustrated in Fig. 1.7. Using this concept as a foundation, we can iteratively focus variables using different thresholds for the variable importance in projection as illustrated in Fig. 1.8. Figure 1.8 also shows that despite the challenges of the dataset, 8,804 variables and only 23 data records, a small set of driving variables can be identified. The identified variable sets can, furthermore, be used as the foundation for polynomial models.

1.2.3 *Fit Regularization—LASSO and SINDy*

Conventionally, the coefficients for a linear model are found using least squares regression which minimizes the norm of the residual error. However, there are other criteria which can be used to determine the coefficients. For instance, ridge (also known as Tikhonov) regularization tries to control the size of the coefficients and mitigate the risks of collinearity in the variables. LASSO (Least Absolute Shrinkage and Selection Operator) regression, meanwhile, assumes that not all of the top-level terms matter. The problem with regularization approaches is two-fold: (1) the proper top-level terms must be present and (2) the proper regularization weight must be applied. McConaghy used this approach with randomly synthesized basis sets with success with problems amenable to linearization [14].

SINDy (Sparse Identification of Nonlinear Dynamical systems) should be mentioned in this space since it follows a similar strategy to identify differential equations for time series modeling. Keijzer also pursued this path [3].

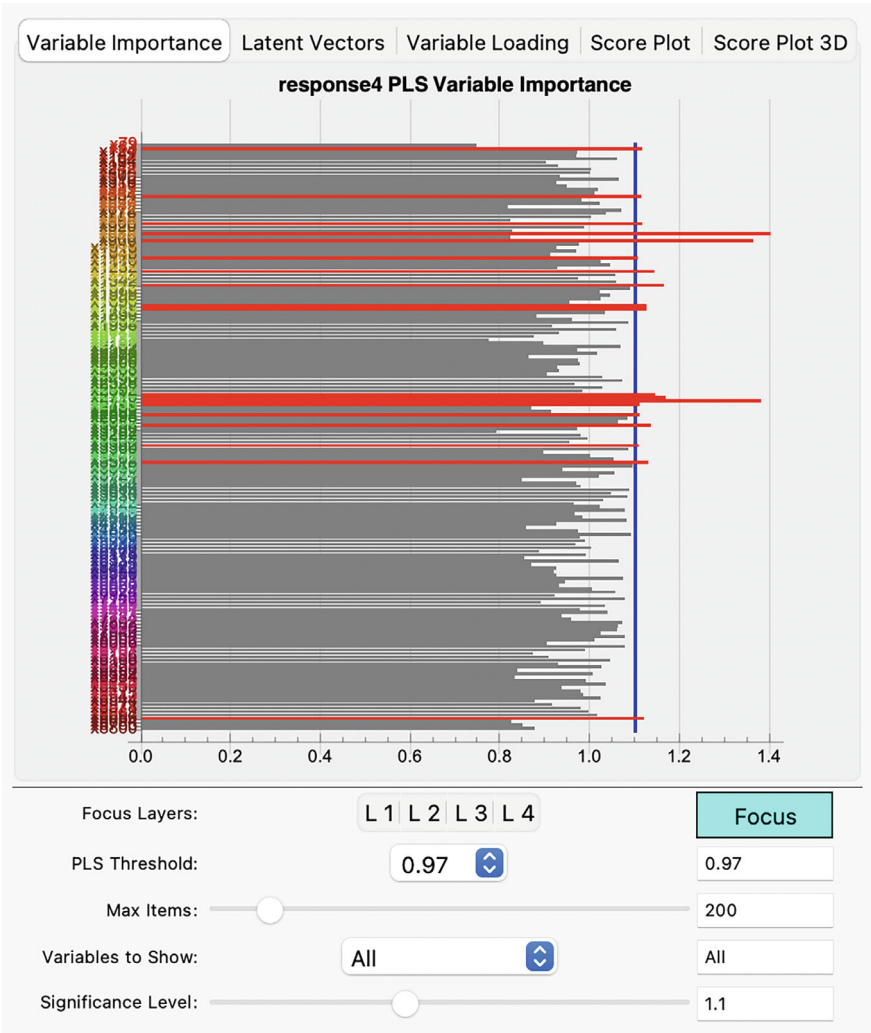


Fig. 1.7 We can iteratively identify dominant variables using PLSR by setting a Significance Level on the variable importance (median variables = 1) and focusing. Here we have started with 8,812 input variables

1.2.4 Symbolic Regression

We will address symbolic regression in more detail later; however, the notable difference between symbolic regression and the linear approaches is that, they define the model form a priori, while symbolic regression lets the data define the model form as well as the variables used. We will assume a good model has two attributes: simple

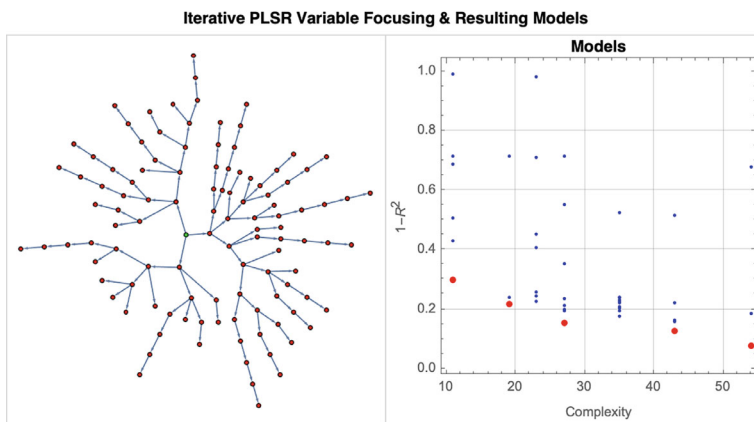


Fig. 1.8 PLSR handles wide data sets by identifying the most significant inputs as projected onto the dominant decomposition orthogonal vectors. Here we have applied that process iteratively from the original 8804 input variables (green root node) using significance levels of 1.0, 1.2 and 1.4. To produce variable sets of fewer than four variables since we only have 23 data records available

and accurate. We have additional assumptions which must be made in terms of the allowable operators and how long we should allocate to the evolutionary search.

Note, that the appropriate level of model complexity is a system attribute and, as a result is an emergent property rather than something that can be defined a priori.

The diverse model forms developed during symbolic regression also offers insight possibilities since we can mine the models which best balance the accuracy-complexity trade-off for insight relative to variable usefulness, combinations and associations—the model search is an optionizer as well as an optimizer. The diversity of developed model forms can also be used to define ensembles whose predictions have an associated trust metric.

1.3 Managing Model Risk

1.3.1 Verify the Data

Aristotle maintained that women have fewer teeth than men; although he was twice married, it never occurred to him to verify this statement by examining his wives' mouths.

—Bertrand Russell (1872–1970), *Impact of Science on Society* (1952), Chap. 1

Garbage in. Garbage out. If we don't start with quality data, our chances of achieving a quality model are minimal. Towards that end, we need to verify that missing data is properly flagged and not masquerading as a numeric. Ideally, we have uniform coverage across the input variable space with gaps in the coverage implying interstitial gaps that must be spanned by the models. Quantized data infers a similar risk since

models are only locked down at observation points and are free to wander between those references.

Additionally, we want to look at the distribution of the inputs and responses since exponential behaviors there tell us that we want to include appropriate operators or log-scale the response. We shall see later that such can convert a hard problem into an easy one.

1.3.2 *Validate the Assumptions*

Trust but Verify
—Russian Proverb

We have postulated that achieving a good model is dependent upon making good assumptions. Figure 1.9 summarizes some of the ways to build confidence in developed models. Obviously, if we have physical insight into the model form, we can prefer structures aligned with the anticipated behavior. Failing that, we can rely upon statistical metrics like ANOVA or fit regularization to trim models to those terms which meet classical criteria.

If we do not have the concept of complexity control, partitioning the data into independent train-test-validate data sets is generally used. The downside of this strategy is the developed models are intentionally under-informed since they do not have access to the full data set. There is also the issue of crafting the partitions so each represents the system dynamics to the extent possible. This scheme breaks down if data is sparse with a cross-validation strategy generally the fallback. This attempts to confirm the model structure by cycling through data subsets with statistics on the held-out data providing the metric for the assessment. The problem with this approach is that if we fit the model to the data subset or use a correlation metric, the model shifts. Further, these methods aren't applicable in scenarios where data acquisition is prohibitively expensive. In those scenarios, it is necessary to extract as much information as possible from those limited samples.

Complexity control should be a preferred strategy. For some approaches like support vector machines or stacked analytic nets, model complexity and response smoothness is baked into the formulation. For symbolic regression, we have a number of mechanisms: expression complexity, model dimensionality and nonlinearity estimation. These will be discussed later.

Ensembles of diverse models also provide a complexity control behavior if the median is used for prediction, excursions of any individual model will be mitigated and the divergence provides a trust metric during prediction.

1.4 Symbolic Regression

Your assumptions are your windows on the world. Scrub them off every once in a while, or the light won't come in.
—Isaac Asimov

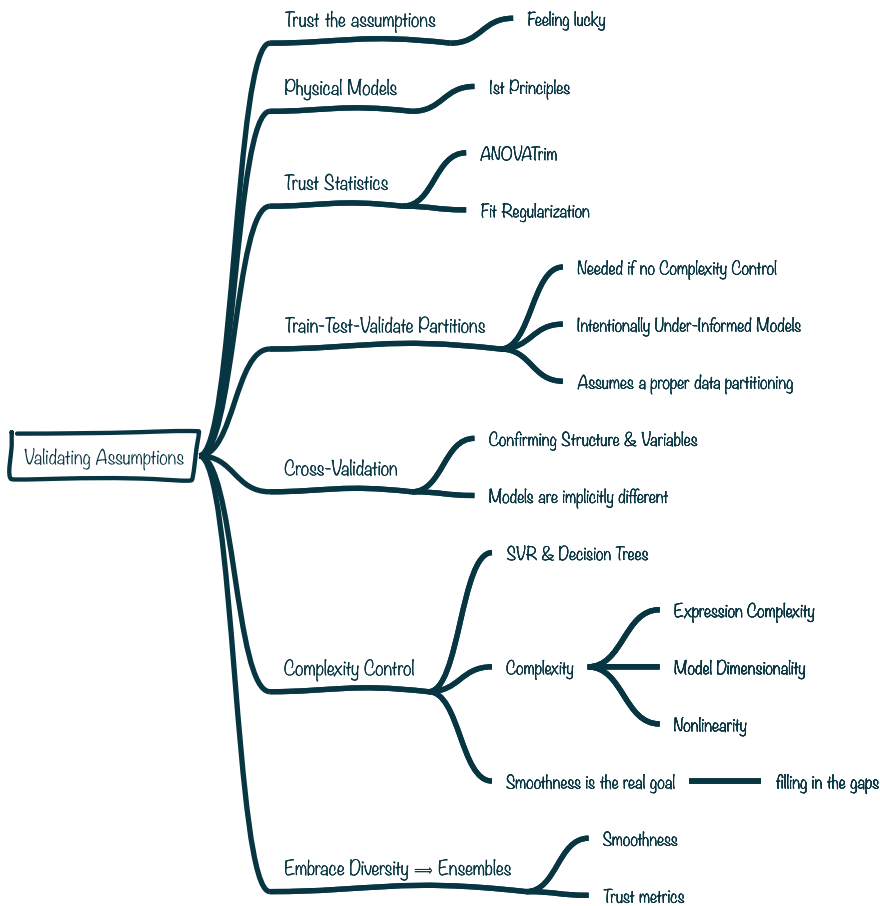


Fig. 1.9 The true goal of validating assumptions is likely smoothness in the interstitial regions

Turning our attention to representation issues as applied to genetic programming and symbolic regression, we have tried to turn a critical eye towards convention and dogma using a variety of toy problems in our exploration.

1.4.1 Performance Measures

If you want to grow a better potato, FIRST you have to be able to measure the potato
—Henri Greenbauer

Continuing with our assumption that a good model is both simple and accurate, let us consider options for measuring those. Explicitly presenting the trade-off of complexity versus accuracy to the user has the benefit of slowing them down from indiscriminately chasing accuracy without respect to the model complexity and fidelity of the data.

1.4.1.1 Accuracy

Don't worry about people stealing an idea. If it's original, you will have to ram it down their throats.

—Howard Aiken (1900–1973)

From a practitioner standpoint, using correlation has a huge advantage over RMSE (root-mean-square-error) in that it is a shape-matching metric. Since we can easily discover scaling and translation coefficients to align the models *post facto*. If we use a norm-based accuracy metric, not only do we have to discover the functional form but the scaling and translation coefficients—which is a much harder problem for evolution. Keijzer published this for symbolic regression in the early 2000's [8]; however, it still seems novel to some [4].

We also have the option of searching directly for the model or to identify basis sets. The former tends to favor multiplicative models whereas evolving basis sets implicitly favors additive models. For many/most industrial data modeling applications, the computational overhead associated with fitting the bases to the data at each stage of search is generally cost-effective. The downside of searching for banks of linearizing transforms is that additive models tend to be harder to interpret and metavariables (aka, small variable combinations and transforms) are not as interesting. For DataModeler, the default behavior for the the basis set search is to cap the number of bases for any give search at a random number between three and ten (Fig. 1.10).

1.4.1.2 Simplicity

Seek simplicity, and distrust it.

—Alfred North Whitehead (1861–1947)

There are many possible metrics for model complexity (Fig. 1.11). Simple measures such as depth, nodes or leaves have the problem that they are coarse. Our preferred metric is the visitation length (nodes travel from the root node to all possible nodes—this is equivalent to the sum of the sum of all possible subtrees) which gives room for interesting models to slot. This metric has a bias for shallower trees for a given number of nodes. Model dimensionality (aka, number of variables) is also something we prefer so it can be included as a quality metric.

Minimizing model nonlinearity is our true goal. The attraction of the visitation length criteria is that it is computationally very efficient. Heuristics such as augmenting that with function weighting [9] don't really address that as well as needed. A

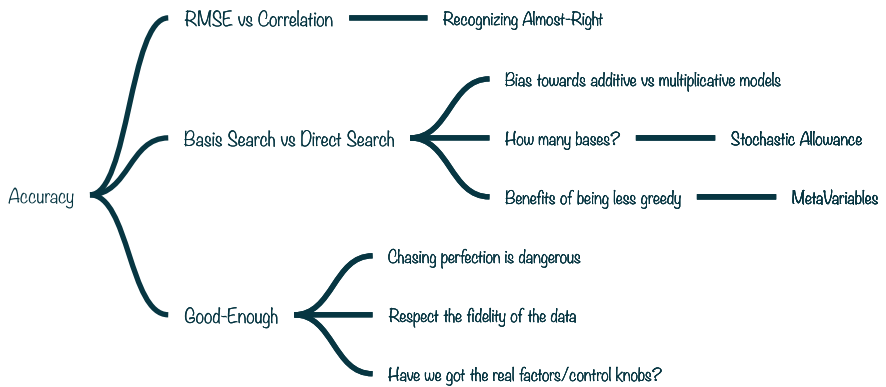


Fig. 1.10 The metric chosen for accuracy can have a huge impact on the efficiency and efficacy of the model search



Fig. 1.11 There are many attributes to model simplicity; however, our true goal is to control the behavior in the interstitial gaps

more attractive approach is to dither synthetic data around observed records to try to capture the nonlinearities within the unobserved interstitial gaps [1].

As mentioned earlier, the proper model complexity is an emergent property. As a result, it is better to adopt a multi-objective approach to model selection than to try to use a parsimony pressure adjustment for single-objective search since parsimony pressure enforces a relationship between accuracy and complexity which must be predefined.

Interval arithmetic [8] is a way to ensure that there are no singularities within the hypercube of model parameter space. The plus is that interval arithmetic is very fast to evaluate with the downside that it is conservative and will detect singularities which are not real. The other issue is the implicit assumption that singularities are not appropriate. For example, if tray temperatures in a distillation column move in concert, it does not matter if we have a singularity if $T_1 < T_2$ since that condition can not be reached in practice even though the ranges overlap. However, if we have quantized data, the conservative constraint may still be desirable.

1.4.2 Function Sets

Nothing is as simple as we hope it will be.

—Jim Horning

The primordial vat of symbolic regression starts with the data variables, a few constants and operators which combine those into models that, hopefully, capture the observed response behavior. Hence, a key assumption of symbolic regression is which operators to place in the primordial soup. There is a downside of including too many or inappropriate ones since they can slow down the evolutionary pressure (since the spurious operators must be considered and suppressed) and some operators can create models which are twitchy and hard to interpret.

Conversely, unless we are certain that the world is a polynomial, we want to include a diverse set of operators. To mitigate the bias towards polynomial forms, DataModeler has introduced a new default functional building block, the continued fraction $CF(x_1, x_2, x_3, x_4) = \frac{x_4}{x_2 + \frac{x_3}{x_1}}$. The continued fraction is more stable than it looks visually and behaves opposite of polynomials since in many cases the continued fractions are bounded as they approach infinity, whereas polynomials explode when extrapolating towards infinity as is demonstrated in Figs. 1.12 and 1.13. The new genetic building block has a model complexity of nine whereas the expansion would have a genome complexity of 32. Note that such structures are commonly evolved during model search so what we have done is leveled the playing field relative to polynomials and square roots.

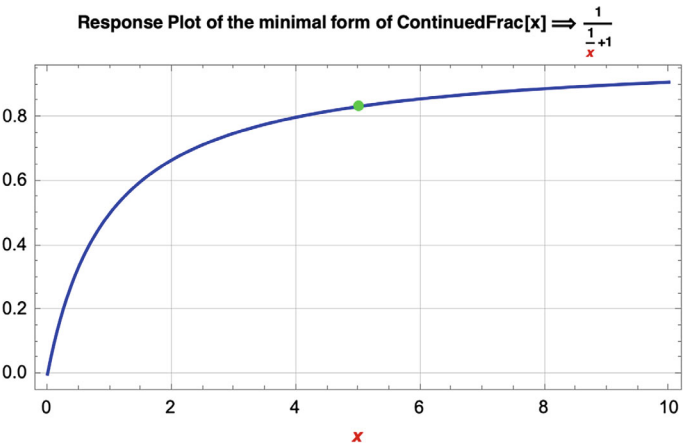


Fig. 1.12 Although visually complicated in its expansion, the continued fraction is actually well-behaved and gives symbolic regression additional flexibility in model discovery

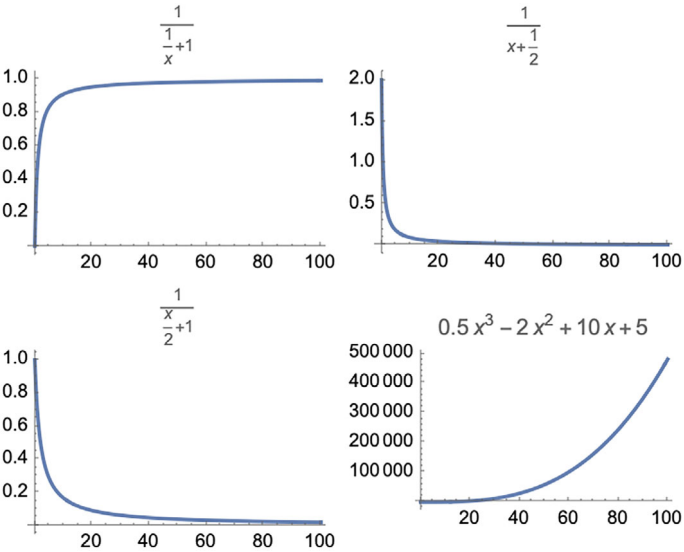


Fig. 1.13 The continued fraction offers stability when extrapolating due to bounds, unlike polynomials which explode when extrapolating

1.4.3 Data Handling

It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.
—Sir Arthur Conan Doyle (1859–1930)

The other major ingredient of the evolutionary soup is the data. As mentioned earlier, we need to ensure that the data is of good quality. As part of that examination, we can identify that it might be worthwhile to transform the inputs or response. The big issue with data scaling is that for multivariate responses the scaling breaks the variable coupling. Hence, in general, it is better to provide appropriate operators (e.g., Log or Exp) or explicitly specify metavariables with transforms or couplings of inputs.

The problem with real-world data sets is that ground truth is not known. Hence, for this chapter we modeled the 100 expressions in the Feynman Equation set [17]. Seventy five were exactly solved using default settings within five minutes (median of three seconds and mean of 17) and another five by tweaking some parameters. However, the more important insight was gained by looking at problems which were difficult. Equation 43 was problematic with exact solutions discovered with the allotted five minutes about half the time; however, if we log-transform the response, exact solutions would reliably be uncovered within 20–30 seconds on a MacBook Pro as is shown in Fig. 1.14.

The lesson learned is that future versions of DataModeler may include automatic response transforms as well as input metavariables.

Data sampling is also something that can affect model discovery. The foundation of OrdinalGP [16] is that a crappy model will be crappy against a subset of the data so, if we are faced with large numbers of data records, we can accelerate the model discovery by evaluating models against (changing) smaller data subsets and, thereby, get through more evolutionary cycles if we are compute-bound by the size of the data set. A side-effect of this is that we reward model generality since good models survive against the changing fitness landscape—hence, an analogy of cross-validation on steroids would be valid.

BalancedGP extends this by bracketing the response into equal increment range bins and sampling from those. Korns [10] applied this to financial data. The benefit is that when data is lumpy (e.g., a system in closed-loop control) we are rewarding a global model capturing the extremes rather than the muddling middle.

Active Learning/Active DOE [5, 6, 12] can be used when data acquisition is expensive to focus on regions of parameter space which provide the most insight and predictive possibilities.

The net is that relatively minor adjustments to the data used in model discovery can have significant impact.

1.5 Representation

The plural of anecdote is not statistics.
—Glenn Fleishman

There are a number of genetic programming representations: trees, stacks, linear, Cartesian, etc. The original tree representation from Koza [13] was plagued by expression bloat during evolutionary search. Most of the other representations were

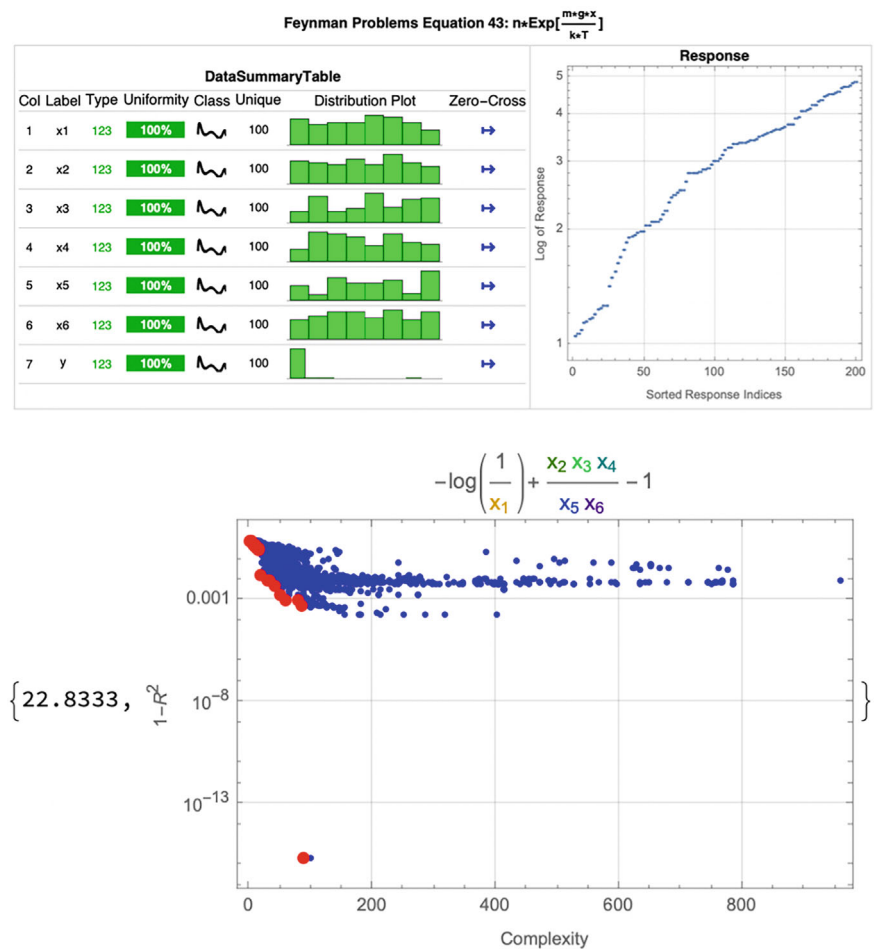


Fig. 1.14 Log-scaling the response for the Feynman Equation 43 converted it from being difficult to trivial

introduced to mitigate that concern. However, with the introduction of the multi-objective ParetoTourney selection strategy [11], bloat has been a solved problem for two decades now. Stacks (PushGP, StackGP, etc.), however, are attractive because they support variable typing much more gracefully than trees and have advantages for evolving decision trees and classifiers.

A downside of stacks and linear genomes is that transferring functional subcomponents is difficult and the genetic operators are mostly focused on mutation. There is a dogma that mutation is the most important diversity operator in genetic programming. However, let us consider the Katya Ball [19] which is a five dimensional problem having an exact solution of the form,

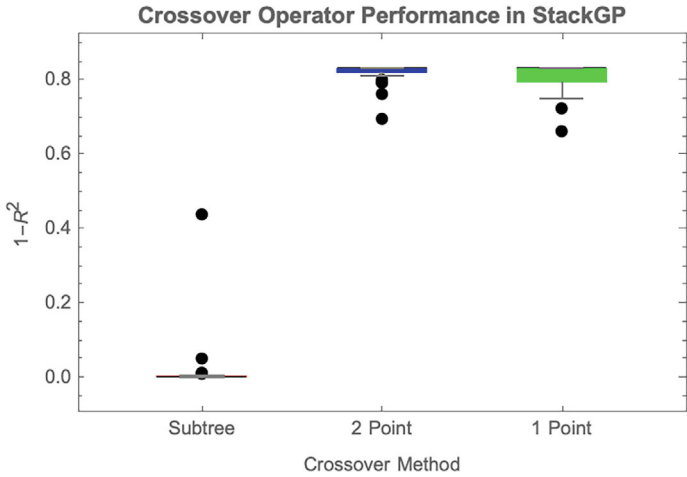


Fig. 1.15 Implementing crossover in StackGP enabled it to solve the Katya Ball problem

$$\frac{10}{(x_1 - 3)^2 + (x_2 - 3)^2 + (x_3 - 3)^2 + (x_4 - 3)^2 + (x_5 - 3)^2 + 5} \tag{1.1}$$

We have 1000 observations over the common data variable of [0.05, 6.05]. Three minute searches in DataModeler searching for the model directly would discover the exact model approximately half of the time. However, if we turned off crossover, then the model quality would cap out at about 0.6 R^2 . Using this insight, a crossover operator was implemented in StackGP. As shown in Fig. 1.15, this led to StackGP shifting from failure to success.

The lesson here is that the genetic representation can make some things easier and others harder. Just because something is hard in a particular representation doesn't mean it isn't worth doing.

1.5.1 Evolution

It ain't what you don't know that gets you into trouble, it's what you know for sure that just ain't so.
—Mark Twain

Evolution is remarkably robust when it comes to search parameters. Generally, if we can maintain sufficient diversity a quality result can be discovered. However, we can stack the deck in our favor by judicious design decisions. We can also stack the deck in our favor by confirming the cultural dogma is appropriate rather than just inertia. Multi-objective search has changed the game. Dogma 30 years ago was that symbolic regression should be done using massive population sizes (e.g., 10,000 models) whereas with a multi-objective search considering complexity and accuracy (and,

sometimes, model age) sizes in the realm of 300 models are provably better since bloat is no longer a concern. The ParetoTourney selection strategy where breeding populations are chosen by selecting a subset of the population (typically ten percent) and rewarding those who define the Pareto front of that subset will focus the evolutionary energy on the knee of the overall Pareto front (aka, those models which best capture the complexity-accuracy trade-off) as well as maintaining population diversity.

As seen with the Katya Ball problem, evolutionary operators affect the ability to discover solutions. Those capabilities also ripple back into other settings. To illustrate, the optimal ParetoTournament size for StackGP shifted from five to thirty models (out of a population of 300) when the crossover operator was implemented.

Many computer languages—most notably Python—have problems with operator overloading and, as a result, restrict addition and multiplication to be pairwise operations. This restriction on arity then ripples into the reachability of model search as illustrated by comparing the results in Figs. 1.16 and 1.17.

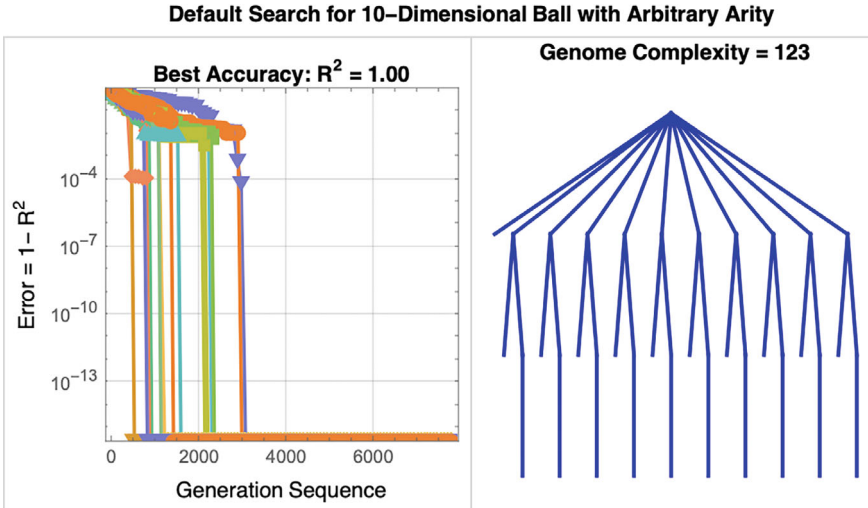


Fig. 1.16 Here we search for a model of the form $f(x) = x_1^2 + x_2^2 + x_{10}^2 + \dots$, i.e., a 10-dimensional ball. No tuning was done other than to increase the BasisSetLimit to 20 instead of the default random setting between three and ten which would kill off models with more than that many terms at the top level. Two minutes of search time was allotted on a 16 core MacBook Pro. The curves on the left show the maximum accuracy achieved over time for each of the 16 parallel searches. The tree plot on the right shows the genetic structure of the last model in the overall Pareto front

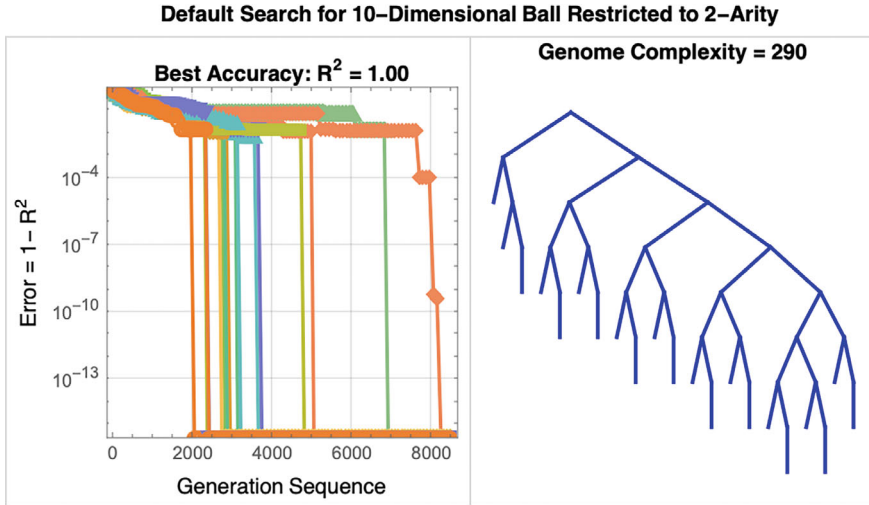


Fig. 1.17 Here we restrict the arity for functions to a maximum of two so that instead of having a direct form $\Sigma(a, b, c, d)$ we have to discover a form of $\Sigma(\Sigma(a, b), \Sigma(c, d))$ or an equivalent. As we can see, unnecessarily restricting addition and multiplication to be pairwise slows down the search and increases the genetic code complexity of models

1.5.2 Selection

Lexicase has become dogma in recent years, replacing tournament selection as the favored selection method in GP. The appeal of lexicase is that since it doesn't utilize summary statistics, specialists or individuals that perform well on individual training cases can survive long enough for their potential to be realized [7]. Compared to other selection methods reliant on summary statistics, the importance of a small subset of training data can be missed simply due to the insignificant impact they have on the fitness metrics. It is important to consider though, that by dropping summary statistics, we lose the global picture thus certain benefits of considering the whole picture. Within the scope of symbolic regression, we specifically miss out on the proven advantages of using correlation as a fitness function which allows us to find solutions that have the correct shape but may have the incorrect scaling and translation coefficients that would otherwise be thrown out by Lexicase and even other global fitness metrics that rely only on distance (RMSE).

We compared lexicase selection and Pareto tournament selection using the Katya Unwrapped Ball problem in both generation constrained and runtime constrained settings (Fig. 1.18). Since this is a regression problem, we utilized epsilon lexicase selection. As well, to see if correlation could be utilized by lexicase in an effective way, we implemented a variant of batch epsilon lexicase selection where correlation was computed for each batch of training cases. We compared two variants of Pareto tournament selection. One using correlation and one using RMSE.

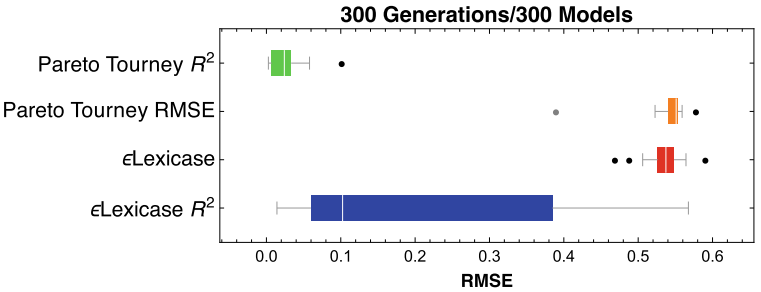


Fig. 1.18 Results of the generation constrained comparison of selection methods on the test data for the Unwrapped Katya Ball Problem

The results comparing the generation constrained approaches on the test set is shown in Fig. 1.21. The results here show that in a generation constrained scenario, Pareto tournament selection using correlation performs best, followed by batch epsilon lexicase using correlation, although the distribution is very wide for the batch epsilon lexicase approach. Both standard epsilon lexicase selection and Pareto tournament selection with RMSE performed poorly, with epsilon lexicase performing marginally better than Pareto tournaments with RMSE. One important factor to note is that when comparing the runtimes of the different approaches we can see that the Pareto tournament approaches perform significantly faster than the lexicase approaches as shown in Fig. 1.19. Recent works within the scope of lexicase selection have already highlighted the significant computational costs of standard lexicase and have developed new approaches which have significantly improved runtimes, specifically the plexicase and DALex variants [2, 15].

Although generation constrained comparisons can be interesting for theoretical analysis, we contend that only runtime constrained comparisons are important in application. For the runtime constrained approach we also included a modified Pareto tournament selection which instead of using complexity and accuracy as the two

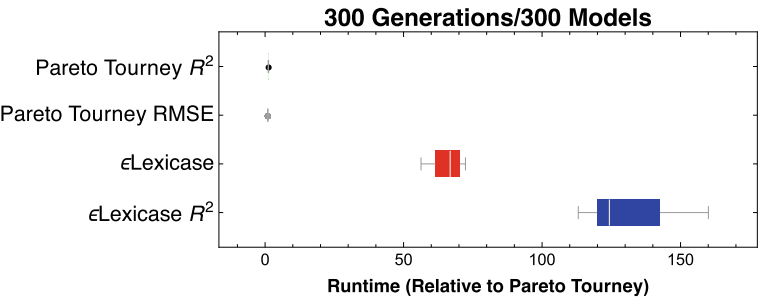


Fig. 1.19 Runtimes of the section methods in the generation constrained comparison for the Unwrapped Katya Ball Problem

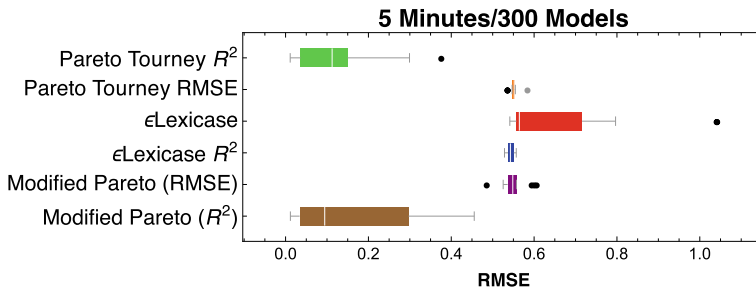


Fig. 1.20 Results of the runtime constrained comparison of selection methods on the test data for the Unwrapped Katya Ball Problem

objectives, it randomly chooses two subsets of data and the accuracy on those two subsets are used to identify the Pareto front of each tournament. The results are shown in Fig. 1.20 and demonstrate that in a short time-span of 5 min, the Pareto tournament approaches with correlation are able to identify high-quality models. The modified Pareto tournament approach has a slightly lower median error but is less consistent, displaying a wider distribution. Epsilon lexicase selection has the highest median error with a relatively wide distribution. When using correlation with lexicase we observe a much smaller distribution and get slightly better accuracy than standard epsilon lexicase selection.

The results demonstrate that Pareto tournament selection is computationally very efficient at optimizing for multiple objectives, orders of magnitude faster than lexicase selection, whether they are competing objectives such as complexity versus accuracy or different partitions within the data as in the modified Pareto approach. In scenarios where there are constraints on time or computation (most real-world problems), these runtime advantages can provide significant benefits allowing deeper evolution or many independent evolutions to occur to improve the coverage of the search space. This comparison also highlights that for symbolic regression problems, there is a significant benefit in considering summary statistics, specifically correlation, since it allows for improved visibility of solutions in the search space. We plan to further analyse the performance of Pareto tournament selection in multi-objective problems outside the scope of symbolic regression to see if these benefits generalize.

1.6 Symbolic Regression Analysis and Model Selection

To have doubted one's own first principles is the mark of a civilized man.
—Oliver Wendell Holmes Jr. (1841–1935)

When all of the searching is done, we have (hopefully) a large number of models which are acceptably simple and acceptably accurate. This candidate set can be mined

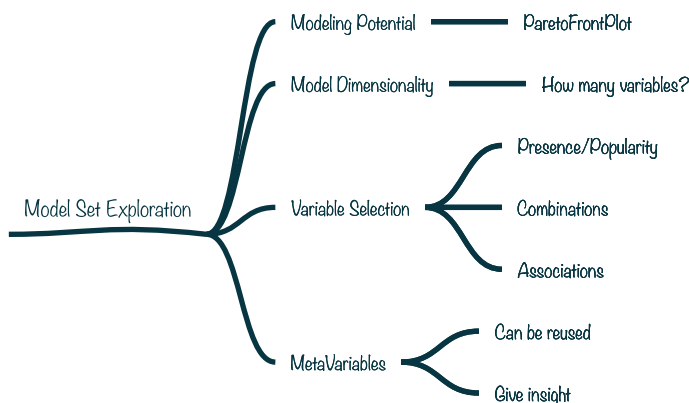


Fig. 1.21 A lot can be learned from a candidate set of models which are both simple and accurate. This can focus subsequent model development as well as provide insight into mechanisms and variable couplings

to determine the number of variables required for a deployed models as well as to focus on variable subsets for further exploration and model development.

For model deployment, we recommend the use of ensembles of diverse models from the accurate-and-simple candidates. Currently, diversity is defined using the lack of correlation of model error residuals with an attempt to over-weight the knee of the candidate models. However, that is an assumption that needs further investigation.

1.7 Conclusions and Summary

No one really listens to anyone else, and if you try it for a while you'll see why.

—Mignon McLaughlin

Opening with the premise that modeling assumptions constrain the reachability of desired solutions, we argue that a good model consolidates energy into a more parsimonious representation and, furthermore, that the purpose of a model is to predict in the interstitial gaps between observations.

We then looked at symbolic regression within the overall data modeling tool suite as well as with other algebraic modeling techniques which also generate implicitly transparent model forms and the alternate variable focusing approaches of PLSR and LASSO.

Since validity of assumptions are foundational we emphasized the importance of data validation and advocated for the role of explicit complexity control as part of multi-objective model search as well as celebrate reach the two decade anniversary of bloat being an issue for tree-based symbolic regression as well as emphasize the benefits of using correlation rather than residual error as an accuracy metric.

Since nature does not know it is a polynomial, we introduce a new operator into the function pattern suite. The continued fraction forms can be naturally evolved; however, having an explicit form reduces the complexity penalty imposed without a functional form.

We showed that appropriate data transforms can convert hard modeling problems into easy ones and emphasized that a holistic approach in the modeling process is critical to success.

We also advocated for the role of crossover and preservation of functional subsections and demonstrated the advantages of such even for representations which do not naturally support such genetic transfer.

We demonstrated that Pareto tournament selection is efficient for optimizing multiple objectives and identified that lexicase selection may be missing out by completely ignoring summary statistics of the training data.

Finally, we demonstrated the benefits of arbitrary arity in model search and not restricting multiplication and addition to pairwise operations.

The broad conclusion is that there is value in confirming that dogma and opinions are grounded in reality.

References

1. Bakurov, I., Haut, N., Banzhaf, W.: Sharpness-aware minimization in genetic programming. arXiv (2024)
2. Ding, L., Pantridge, E., Spector, L.: Probabilistic lexicase selection. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'23, pp. 1073–1081. Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3583131.3590375>
3. Gaucel, S., Keijzer, M., Lutton, E., Tonda, A.: Learning dynamical systems using standard symbolic regression. In: Genetic Programming: 17th European Conference, EuroGP 2014, Granada, Spain, April 23–25, 2014, Revised Selected Papers 17, pp. 25–36. Springer (2014)
4. Haut, N., Banzhaf, W., Punch, B.: Correlation Versus RMSE Loss Functions in Symbolic Regression Tasks, pp. 31–55. Springer Nature, Singapore (2023). https://doi.org/10.1007/978-981-19-8460-0_2
5. Haut, N., Banzhaf, W., Punch, B., Colbry, D.: Accelerating image analysis research with active learning techniques in genetic programming. In: Hu, T., Ofria, C., Trujillo, L., Winkler, S. (eds.) Genetic Programming Theory and Practice XX. Genetic and Evolutionary Computation. Michigan State University, USA (2023) (Forthcoming)
6. Haut, N., Punch, B., Banzhaf, W.: Active learning informs symbolic regression model development in genetic programming. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation, GECCO'23 Companion, pp. 587–590. Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3583133.3590577>
7. Helmuth, T., Spector, L., Matheson, J.: Solving uncompromising problems with lexicase selection. *IEEE Trans. Evol. Comput.* **19**(5), 630–643 (2015). <https://doi.org/10.1109/TEVC.2014.2362729>
8. Keijzer, M.: Improving symbolic regression with interval arithmetic and linear scaling. In: European Conference on Genetic Programming, pp. 70–82. Springer (2003)

9. Kommenda, M., Beham, A., Affenzeller, M., Kronberger, G.: Complexity measures for multi-objective symbolic regression. In: Computer Aided Systems Theory–EUROCAST 2015: 15th International Conference, Las Palmas de Gran Canaria, Spain, February 8–13, 2015, Revised Selected Papers 15, pp. 409–416. Springer (2015)
10. Korns, M.F.: Large-scale, time-constrained symbolic regression. In: Genetic Programming Theory and Practice IV, pp. 299–314 (2007)
11. Kotanchek, M., Smits, G., Vladislavleva, E.: Pursuing the pareto paradigm: tournaments, algorithm variations and ordinal optimization. In: Genetic Programming Theory and Practice IV, pp. 167–185 (2007)
12. Kotanchek, M., Smits, G., Vladislavleva, E.: Exploiting trustable models via Pareto GP for targeted data collection. In: Riolo, R., Soule, T., Worzel, B. (eds.) Genetic Programming Theory and Practice VI, pp. 145–162. Springer (2009)
13. Koza, J.R.: Genetic Programming: On the Programming of Computers By Means Of Natural Selection. MIT Press, Cambridge, MA (1992)
14. McConaghy, T.: FFX: Fast, scalable, deterministic symbolic regression technology. In: Genetic Programming Theory and Practice IX, pp. 235–260 (2011)
15. Ni, A., Ding, L., Spector, L.: Dalex: Lexicase-like selection via diverse aggregation. In: Giacobini, M., Xue, B., Manzoni, L. (eds.) Genetic Programming, pp. 90–107. Springer Nature Switzerland, Cham (2024)
16. Smits, G., Vladislavleva, E.: Ordinal pareto genetic programming. In: 2006 IEEE International Conference on Evolutionary Computation, pp. 3114–3120. IEEE (2006)
17. Tegmark, M.: Welcome to the Feynman symbolic regression database! <https://space.mit.edu/home/tegmark/aifeynman.html>
18. Trujillo, L., Z-Flores, E., Juárez-Smith, P.S., Legrand, P., Silva, S., Castelli, M., Vanneschi, L., Schütze, O., Muñoz, L.: Local search is underused in genetic programming. In: Genetic Programming Theory and Practice XIV, pp. 119–137 (2018)
19. Vladislavleva, E.J., Smits, G.F., Den Hertog, D.: Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *IEEE Trans. Evol. Comput.* **13**(2), 333–349 (2008)
20. Wold, H., Lyttkens, E.: Nonlinear iterative partial least squares (NIPALS) estimation procedures. *Bull. Int. Stat. Inst.* **43**(1), 29–47 (1969)

Chapter 2

EvoFeat: Genetic Programming-Based Feature Engineering Approach to Tabular Data Classification



Hengzhe Zhang, Qi Chen, Bing Xue, Yan Wang, Aimin Zhou,
and Mengjie Zhang

Abstract In recent years, in the context of tabular data classification, the emergence of the transformer architecture has led to deep learning methods yielding better results than conventional tree-based models. Most of these findings attribute the success of deep learning to the expressive feature construction capabilities of neural networks. Nonetheless, in real-world practice, manually designed high-order features using traditional machine learning methods are still widely used because features based on neural networks can be prone to overfitting. In this paper, we propose a genetic programming-based feature engineering algorithm to automate the feature construction process through trial and improvement. Importantly, genetic programming provides an opportunity to optimize symbolic models, which gradient-based methods often find hard to optimize. On a large-scale classification benchmark involving 130 datasets, the experimental results demonstrate that the proposed method outperforms existing, fine-tuned state-of-the-art tree-based and deep-learning-based classification algorithms.

H. Zhang · Q. Chen · B. Xue · M. Zhang (✉)

The Centre for Data Science and Artificial Intelligence & School of Engineering and Computer Science, Victoria University of Wellington, PO Box 600, Wellington 6140, New Zealand
e-mail: mengjie.zhang@ecs.vuw.ac.nz

H. Zhang

e-mail: hengzhe.zhang@ecs.vuw.ac.nz

Q. Chen

e-mail: qi.chen@ecs.vuw.ac.nz

B. Xue

e-mail: bing.xue@ecs.vuw.ac.nz

Y. Wang

School of Computing, Macquarie University, Sydney, NSW 2109, Australia

e-mail: yan.wang@mq.edu.au

A. Zhou

Shanghai Institute of AI for Education, School of Computer Science and Technology, East China Normal University, Shanghai 200062, China

e-mail: amzhou@cs.ecnu.edu.cn

2.1 Introduction

Tabular data is a typical way of storing information in computer systems, and tabular data learning techniques have been widely employed in recommendation systems [37] and advertising [39] to automate decision-making processes. Formally, these techniques capture the association between explanatory variables $\{x_1, \dots, x_m\}$ and a response variable y within a given dataset $\{(\{x_1^1, \dots, x_m^1\}, y^1), \dots, (\{x_1^n, \dots, x_m^n\}, y^n)\}$, where n represents the number of data instances.

Various machine learning methods, such as decision trees, linear models, and support vector machines, can address the tabular data learning problem. However, these algorithms have limited capabilities in fully capturing the complex relationships between explanatory variables and response variables in real-world tabular data. For instance, linear models assume a linear relationship between the explanatory and response variables, while decision trees posit that an axis-parallel decision boundary can effectively divide distinct categories of samples. Given that these assumptions do not always hold in the real world, machine learning practitioners often employ feature engineering techniques to construct a set of k high-order features $\{\{\phi_1^1, \dots, \phi_k^1\}, \dots, \{\phi_1^n, \dots, \phi_k^n\}\}$, enhancing the learning capabilities of existing algorithms.

Traditional feature engineering strategies include manually designed features, dimensionality reduction methods [7], and kernel methods [1]. Despite their utility, these approaches have inherent limitations. Manually designing features is often highly effective but labor-intensive, kernel methods are difficult to integrate with tree-based learning methods, and dimensionality reduction algorithms often fail to enhance state-of-the-art learning algorithms, particularly unsupervised ones [18]. The pursuit of optimal predictive accuracy in tabular data classification has led to significant advancements in machine learning methodologies. Among these, ensemble learning methods utilizing decision trees and logistic regression have emerged as powerful tools [4], celebrated for their ability to produce stable and accurate predictions across diverse domains. Despite these strides, the performance of such ensembles often hinges on the quality of feature engineering, a step that remains manually intensive and heuristic-driven.

With the success of deep learning, there is growing interest in using neural networks to automatically build high-order features [19, 33]. While neural network-based tabular learning algorithms have demonstrated superior predictive accuracy over some tree-based ensemble models [2], most of these findings are primarily derived from datasets with a large number of training samples and homogeneous features. The superiority of feature learning ability in neural network-based deep learning methods over tree-based methods on datasets with heterogeneous features and limited data remains controversial [9]. This paper aims to leverage genetic programming (GP) as a data-efficient deep learning technique and explore the capability of GP-based representation learning on tabular classification tasks.

In real-world scenarios, machine learning engineers typically begin with some manually designed features and then evaluate these features by determining cross-

validation loss and feature importance. The feedback from this evaluation guides engineers in designing better features. To replicate this workflow, we have developed a feature engineering method named the Evolutionary Feature Engineering Automation Tool (EvoFeat),¹ which constructs nonlinear features based on GP to enhance an ensemble of decision trees and logistic regression models, thus achieving better results than conventional ensemble learning methods. GP, an evolutionary algorithm-inspired method, offers a framework for automatically discovering new feature combinations and transformations tailored to the specifics of the data and the models used. By integrating GP-based feature engineering into ensemble learning, we hypothesize that the resultant models will exhibit superior performance on various benchmarks of tabular data classification. The three objectives of our paper are summarized as follows:

- Considering that cross-validation losses and feature importance values are widely used by machine learning engineers during the feature engineering process, we propose an evolutionary feature construction method that leverages both types of information to guide the search process. Specifically, a softmax selection operator is proposed to select beneficial and detrimental features in each feature set for effective crossover in GP.
- Given that the performance of the constructed features is highly dependent on the base learner, we propose a mechanism that allows GP to automatically determine whether to use random decision trees or logistic regression models as base learners.
- We conduct experiments on 130 datasets and compare our method to six traditional machine learning algorithms and four deep learning algorithms. The results demonstrate that deep learning methods are not a panacea for classification tasks.

The remainder of the paper is structured as follows. Section 2.2 introduces related work on feature construction. Section 2.3 discusses the motivation behind our work. Section 2.4 then explains the details of the proposed method, followed by the experimental results provided in Sect. 2.5. Finally, Sect. 2.6 summarizes our paper and recommends some intriguing future avenues.

2.2 Related Work

Evolutionary feature construction methods have received a lot of attention in the evolutionary learning domain. A notable example is multidimensional multiclass genetic programming with multidimensional populations (M3GP) [26], which allows each individual to employ multiple GP trees to construct features that enhance a Mahalanobis distance classifier. Following this, stack-based GP [17] was introduced to naturally support the construction of multiple features by leveraging the multi-output nature of stacking GP. Nevertheless, most of these constructed features are fed into

¹ Source code: <https://github.com/hengzhe-zhang/EvolutionaryForest/blob/master/experiment/methods/EvoFEAT.py>.

simple learners for classification, such as a single decision tree [34] or a single support vector machine [27]. There is still a gap in research on applying evolutionary feature construction methods to enhance state-of-the-art classification algorithms. In recent years, ensemble-based evolutionary feature construction methods have shown promising results in regression [44]. However, feature construction for these tasks differs from that for tabular data classification tasks because domain knowledge for designing genetic operators is different. This difference is illustrated by a recent example of adapting the semantic hoist mutation operator from regression to classification [42]. In this work, transforming evolutionary feature construction techniques from regression to classification requires adapting the loss function and replacing linear regression with logistic regression, with strategies like one-versus-all as base learners to handle multi-class classification.

In addition to evolution-based methods, beam search could also be used to construct high-order features [11, 23, 32]. These methods start with some low-order features and iteratively generate higher-order features based on important low-order features. A meta learner [11], logistic regression accuracy [23], or XGBoost feature importance [32] can be used to determine the importance of each generated feature. However, these methods only use training loss for feature construction, thus lacking an effective mechanism to prevent overfitting, which may restrict their feature construction capabilities.

In the recommendation system domain, deep learning methods have been shown to effectively construct high-order features for tabular data classification tasks [10]. Although feed-forward neural networks (FNNs) are considered universal function approximators [13], their ability to model high-order feature interactions has been questioned [25]. To efficiently construct high-order features, a cross-network is designed in the Deep & Cross Network (DCN) to explicitly model feature interactions [25]. Following this, xDeepFM employs a field-wise feature cross method to effectively [19] model feature interactions in a bit-wise manner. AutoInt [33] takes this further, learning feature interactions using the attention mechanism rather than a simple outer product. Given that useless feature interactions can introduce noise, AutoFIS [20] employs a generalized regularized dual averaging (GRDA) [5] method to remove unimportant feature interactions. With the increasing popularity of the neural network architecture search (NAS) technique, an algorithm known as AutoPI [24] uses the NAS technique to automatically discover better feature interaction components. This method is inspired by differentiable architecture search (DARTS) [21], using the temperature annealing [21] technique to find the best subnet from a supernet using the gradient descent method. However, the effectiveness of these feature cross modules over FNNs is still debatable [37], let alone the lack of a comprehensive study on whether these approaches can outperform XGBoost in classical classification problems [9].

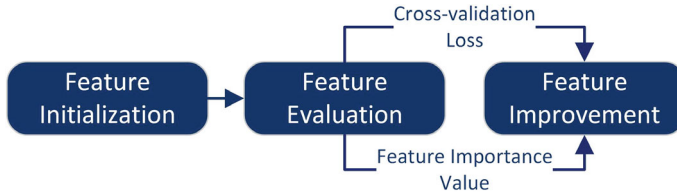


Fig. 2.1 An example of the feature engineering process.

2.3 Preliminaries

First, let us analyze how a machine learning engineer constructs features for a given dataset $\{(\{x_1^1, \dots, x_m^1\}, y^1), \dots, (\{x_1^n, \dots, x_m^n\}, y^n)\}$ in the absence of domain expertise. Figure 2.1 depicts an example of the feature engineering workflow, which is divided into three phases:

- **Feature Initialization:** Initially, a machine learning engineer with domain knowledge may construct features based on the domain knowledge. For those domains without any domain knowledge, they might randomly construct some features $\Phi = \{\phi_1 \dots \phi_k\}$ based on the original features $\{x_1 \dots x_m\}$.
- **Feature Evaluation:** Next, using the cross-validation approach, the generated features $\{\phi_1 \dots \phi_k\}$ are evaluated on the training set with the target learning algorithm. The trained model produces cross-validation losses and feature importance values to help an engineer identify useful features.
- **Feature Improvement:** Finally, the engineer discards some ineffective features and replaces them with new ones derived from important features.

During the feature construction process, the second and third phases are repeated until the engineer is satisfied with the results. Formally, if F represents a search space of machine learning models and $L(f(x), y)$ is the loss function of a model $f \in F$, then the feature construction process aims to find a set of features Φ that minimizes the objective function $\min_{f^* \in F} L(f^*(\Phi(x)), y)$.

In real-world applications, some machine learning engineers prefer to use only feature importance values to detect useful features [32]. However, this approach is risky because feature importance values do not necessarily reflect the generalization performance of the constructed features. In other words, a feature that has a high importance value in an over-fitted model might not necessarily be truly useful. Therefore, a strategy to avoid such misleading information, such as using the cross-validation strategy, is essential. Constructing multiple sets of features simultaneously is a plausible solution. The engineer can then detect the over-fitting issue based on the cross-validation performance of multiple sets of features, gaining valuable insights into their generalization capabilities and making a more informed decision.

The feature engineering paradigm depicted in Fig. 2.1 is general. This approach remains applicable even for a machine learning algorithm that does not provide an internal feature importance report. By using black-box feature importance calculation

methods, such as SHAP [22], it becomes possible to assess feature significance in such cases [36]. Under this paradigm, the most critical challenge is determining how to leverage information based on feature importance values and cross-validation loss to generate better features. Here, we employ tree-based GP (TGP) [15] to do so. TGP is an evolutionary algorithm that focuses on evolving symbolic trees. Its gradient-free nature and variable-length representation make it suitable for generating expressive features based on non-differentiable objective values.

2.4 The Proposed Algorithm

2.4.1 Feature Representation

Unlike traditional evolutionary algorithms, which use an array or vector of real numbers to denote a solution, we employ a set of symbolic trees to represent a set of constructed features. Specifically, each individual has k GP trees, where k is a hyperparameter defined beforehand. For each individual, all symbolic trees $\{\phi_1 \dots \phi_k\}$ in an individual Φ collectively form a k -dimensional new feature space $\mathbb{S}$. Each symbolic tree comprises two parts: non-leaf nodes and leaf nodes. Non-leaf nodes, also known as functions, can transform or combine basic features, such as $\{+, -\}$ or $\{\log, \sin\}$. Leaf nodes, also known as terminals, represent original features, such as $\{x_1, x_2\}$. In addition to a set of symbolic trees, each individual is also accompanied by a base learner, either a decision tree or a linear regression model, used to evaluate the predictive performance of each feature space and to predict unseen data.

2.4.2 Algorithm Framework

The proposed evolutionary algorithm consists of five parts: initialization, evaluation, parent selection, offspring generation, and archive update. The pseudocode for our algorithm is shown in Algorithm 2.1.

Initialization (Line 1): In the initialization stage, the algorithm randomly initializes N individuals $\{\Phi_1 \dots \Phi_N\}$ to form the initial population. Each individual consists of k symbolic trees $\{\phi_1 \dots \phi_k\}$, which are used to construct a k -dimensional feature space.

Evaluation (Lines 2–4, 36–38): At the evaluation stage, we construct training data based on each individual Φ and evaluate it using the associated base learner f of individual Φ . The fitness value of each individual is determined by the five-fold cross-validation loss achieved after applying the constructed features to the base learner f . The importance of each feature $\phi \in \Phi$ is obtained using the average feature importance values across five folds based on five-fold cross-validation.

Parent Selection (Lines 11–18): The selection operator selects promising individuals $\{\Phi_A, \Phi_B\}$ from the population P to generate new feature sets. In this paper, we use the lexicase selection operator [16] to select parent individuals based on the cross-validation losses. For each Φ , EvoFeat uses a softmax selection operator to select pairs of good and bad features $(\phi_{\text{good}}, \phi_{\text{bad}})$ for crossover and a random selection operator to select features for mutation.

Offspring Generation (Lines 19–32): The generation operator generates new individuals based on selected parent features. We use self-competitive based crossover [28] and guided mutation operators to increase search efficiency by leveraging feature importance values.

Archive Update (Lines 6, 39): At the end of each iteration, we dynamically update an archive A to store historically top-performing models based on the performance of archived models A and newly generated models $\{f_1 \dots f_N\}$. The archive update method used in this research is a greedy ensemble selection method called reduce-error pruning [4].

The selection, generation, evaluation, and archive update operators are executed repeatedly until the maximal number of generations is reached. Finally, all base learners in archive A are used to form an ensemble model based on the weights assigned by the ensemble selection method.

2.4.3 *Tree/Feature Initialization*

The proposed evolutionary feature construction method is based on TGP; therefore, we employ a ramped-half-and-half [3] symbolic tree initialization strategy to initialize each tree ϕ . Specifically, half of the symbolic trees are constructed as full trees, while the other half are initialized with a random depth. Additionally, each individual is randomly assigned either a random decision tree (RDT) or a logistic regression model (LR) with equal probability. This approach allows the evolutionary algorithm to determine the optimal base learner to form an ensemble model.

2.4.4 *Tree/Feature Selection*

In EvoFeat, three selection operators are considered. The first is the base learner selection operator, which filters individuals based on the base learner used. The second is the individual selection operator, whose primary purpose is to select two individuals from the population based on their cross-validation loss. The third is the tree/feature selection operator, which selects features from two individuals for crossover and mutation. The evolutionary computation literature offers numerous selection operators, such as Pareto tournament selection [14], age-fitness Pareto selection (AFP) [31], and lexicase selection [16]. In this paper, we employ lexicase

Algorithm 2.1 Evolutionary Feature Engineering Automation Tool (EvoFeat)

Require: Population Size N , Number of Features k , Number of Generations max_gen

Ensure: Archive A

```

1: Randomly initialize  $N$  set of features  $P = \{\Phi_1, \dots, \Phi_N\}$  ▷ Initialization
2: for  $i = 1, \dots, N$  do ▷ Evaluation
3:    $f_i \leftarrow \text{cross-validation}(f_i, \Phi_i)$ 
4: end for
5:  $A \leftarrow \emptyset$ 
6:  $A \leftarrow \text{archive update}(\{f_1, \dots, f_N\}, A)$ 
7:  $gen \leftarrow 0$ 
8: while  $gen < max\_gen$  do ▷ Main loop
9:    $P' \leftarrow \emptyset$ 
10:  for  $i = 1, \dots, N$  do ▷ Selection
11:     $P_{LR}, P_{DT} \leftarrow \text{population division}(P)$ 
12:    if  $rand() < 0.5$  then ▷ Base Learner Selection
13:       $\Phi_A \leftarrow \text{lexicase selection}(P_{LR})$ 
14:       $\Phi_B \leftarrow \text{lexicase selection}(P_{DT})$ 
15:    else
16:       $P' \leftarrow \text{random selection}(P_{LR}, P_{DT})$ 
17:       $\Phi_A, \Phi_B \leftarrow \text{lexicase selection}(P')$ 
18:    end if
19:    for  $j = 1, \dots, \lfloor \frac{k}{2} \rfloor$  do ▷ Crossover
20:      if  $rand() < p_c$  then
21:         $\phi_a, \phi_b \leftarrow \text{softmax selection}(\Phi_A)$ 
22:         $\phi_c, \phi_d \leftarrow \text{softmax selection}(\Phi_B)$ 
23:         $\phi_a \leftarrow \text{self-competitive crossover}(\phi_a, \phi_d)$ 
24:         $\phi_c \leftarrow \text{self-competitive crossover}(\phi_c, \phi_b)$ 
25:      end if
26:    end for
27:    for  $j = 1, \dots, k$  do ▷ Mutation
28:      if  $rand() < p_m$  then
29:         $\phi_a \leftarrow \text{random selection}(\Phi_A)$ 
30:         $\phi_i \leftarrow \text{mutation}(\phi_i)$ 
31:      end if
32:    end for
33:     $P' \leftarrow P' \cup \{\Phi_A, \Phi_B\}$ 
34:  end for
35:   $P \leftarrow P'$ 
36:  for  $k = 1, \dots, N$  do ▷ Evaluation
37:     $f_k \leftarrow \text{cross-validation}(f_k, \Phi_k)$ 
38:  end for
39:   $A \leftarrow \text{archive update}(\{f_1, \dots, f_N\}, A)$  ▷ Archive Update
40:   $gen \leftarrow gen + 1$ 
41: end while
42:

```

selection as the individual selection operator due to its effectiveness in maintaining population diversity, which is beneficial in evolutionary feature construction scenarios [44].

Base Learner Selection: In EvoFeat, we can use methods like a decision tree or logistic regression model as the base learner to evaluate each group of features because they are inexpensive to train. However, decision trees and logistic regression models may not yield comparable predictive results. For instance, since a single decision tree cannot produce a categorical distribution, misclassifying even one example will result in an infinite cross-entropy loss. We borrow an idea from the multitask evolutionary optimization domain [38] to select two parent individuals from a heterogeneous population. First, we divide the entire population into two subgroups: one with individuals using only decision trees as base learners and the other with individuals using only logistic regression models. We then define a hyperparameter called random mating probability $rpm = 0.5$, which indicates the probability of selecting two parents from different subgroups. In the remaining 50% of cases, we select two parents from the same subgroup.

Lexicase Selection: In EvoFeat, lexicase selection is used to select individuals from a population. For a given vector of validation losses, the lexicase selection operator constructs a set of filters based on each loss value across all individuals. Each filter randomly selects one dimension $j \in [1, n]$ for filtering, and the threshold τ_j is dynamically set as Eq. (2.1)

$$\tau_j = \min_i \mathcal{L}_j^i + \varepsilon_j, \quad (2.1)$$

where $\varepsilon_j = \text{median}_i(|\mathcal{L}_j^i - \text{median}_k(\mathcal{L}_j^k)|)$ represents the median absolute deviation. Using this selection operator, we can select a suitable number of parent individuals for crossover and mutation.

Softmax Selection: In EvoFeat, softmax selection is proposed for selecting features from an individual. Since the primary objective of feature engineering is to eliminate low-informative features and construct highly informative ones, the softmax selection operator selects pairs of good and bad features, like $\{(\phi_a, \phi_b), (\phi_c, \phi_d)\}$, to determine which features should be retained and which discarded. The quality of features is determined by feature importance values. For a given set of feature importance values $\{\theta_1 \dots \theta_k\}$ provided by the base learner, we use a softmax function, as shown in Eq. (2.2):

$$P(\theta_i) = \frac{e^{\theta_i/T}}{\sum_{j=1}^k e^{\theta_j/T}}, \quad (2.2)$$

where T represents temperature, to determine the likelihood of selecting each feature θ_i as the good feature, and use a negative softmax function to identify the bad feature. Given the preceding probability distribution, we begin by sampling the good feature ϕ_{good} that provides genetic material proportionate to $P(\theta_i)$. Then we sample the bad feature ϕ_{bad} that needs to be modified proportionate to the probability $P(-\theta_i)$. Based on the selected features $(\phi_{\text{good}}, \phi_{\text{bad}})$, crossover and mutation operators can be used to enhance the quality of features in each individual.

2.4.5 Offspring/Feature Generation

In EvoFeat, there are two operators for transforming two given parent individuals into new individuals, namely a self-competitive crossover operator and a guided mutation operator. An illustrative example of these two operators is given in Fig. 2.2. The self-competitive crossover operator, originally developed to evolve multiple job scheduling rules [28], retains effective rules and modifies less effective ones, mimicking the manual rule design process and making it suitable for our feature construction scenario. Guided mutation, proposed to solve maximum clique problems [45], eliminates less useful variables, which is similarly important for classification problems. Thus, we decide to use these two operators in our algorithm.

Self-competitive Crossover: EvoFeat uses the self-competitive crossover operator to exchange beneficial materials between two parent individuals. Ideally, a feature construction algorithm should be able to transfer important materials from the best feature to the worst feature while keeping the best feature intact. Thus, for a pair of good and bad features ($\phi_{\text{good}}, \phi_{\text{bad}}$), the self-competitive crossover operator selects a subtree from ϕ_{good} at random to replace a subtree in ϕ_{bad} . This ensures that the top-performing features are preserved, while the worst features have a high probability of improvement.

Guided Mutation: In addition to the self-competitive operator, the guided mutation operator [43] is used in EvoFeat to generate new features. Similar to the random mutation operator, it replaces a subtree of a parent individual with a randomly gener-

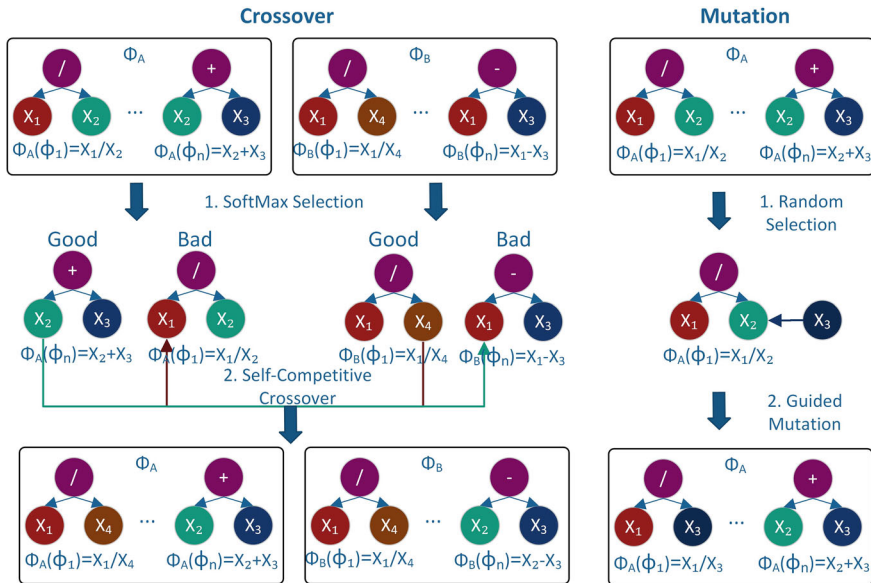


Fig. 2.2 An example of the crossover and mutation operators

ated subtree. However, the guided mutation operator uses a guided probability vector p to determine the likelihood of using each terminal variable in the newly generated subtree.

For each terminal variable x , the guided probability vector $\{p_1 \dots p_m\}$ is defined as the percentage of occurrences of variable x in all features in the archive, weighted by the feature importance of each feature.

- **Decision Tree:** The feature importance values in a decision tree are calculated by counting the total reduction of the Gini impurity contributed by each feature ϕ .
- **Logistic Regression:** In a linear model, feature importance values are obtained by calculating the absolute value of the model coefficients. All features are standardized before constructing the linear model, ensuring that the magnitude of each feature does not influence the model coefficient disproportionately.

2.4.6 Feature Evaluation

For each group of features, feature evaluation is conducted on its associated classification model. To prevent overfitting, we validate the effectiveness of each set of features using a cross-validation approach. For any given training set, we partition it into five subsets/folds, i.e., $\{X_1 \dots X_5\}$. Then, for each round, we use four folds to train a model and the remaining fold to validate the learned model.

Instead of summing all losses into a scalar, we maintain the original loss vector because it contains more detailed and useful information that the selection operator can exploit and utilize. For each training instance, we employ cross entropy as the fitness function. If the problem contains C classes, the cross entropy loss is defined as Eq. (2.3):

$$\sum_{c \in C} p_c * \log(q_c), \quad (2.3)$$

where q_c represents the probability of the model prediction belonging to a certain class c , and p_c is the ground truth probability distribution.

2.5 Experiments

We conduct extensive experiments on both synthetic and real-world datasets to answer the following three questions:

- **Q1:** How does the proposed method compare with traditional machine learning methods (Sect. 2.5.2.1) and deep learning-based methods in terms of performance (Sect. 2.5.2.2)?

- **Q2:** How does dynamically selecting base learners during the evolutionary process compare in performance with pre-selecting them in advance (Sect. 2.5.3.1)?
- **Q3:** Is it more effective to guide the search using both cross-validation losses and feature importance values compared to using cross-validation losses alone (Sect. 2.5.3.2)?

2.5.1 Experimental Settings

Datasets: The datasets are sourced from DIGEN [29]² and the Penn Machine Learning Benchmark (PMLB) [30].³ For the PMLB benchmark, we consider classification datasets with more than 200 instances, but restrict to those where the product of the number of instances and the number of features is $< 10^5$ to avoid exceeding memory limits. DIGEN comprises a diverse set of datasets synthesized using the genetic programming method, while PMLB includes real-world datasets collected from OpenML [35].

Evaluation Protocol: For the evaluation protocol, we randomly sample 80% of the data as the training set to train a model and use the remaining 20% as the test set to calculate the balanced accuracy of the trained model. During training, the fitness value is calculated based on fivefold cross-validation on the training set. To obtain reliable results, we repeat all experiments with 30 random seeds and take the average score as the final score. For all baseline algorithms, we tune them using Heteroscedastic Evolutionary Bayesian Optimization (HEBO) [8]. The details of all hyperparameter search spaces are provided in Table 2.1, and the parameter settings for GP are provided in Table 2.2. Regarding operators, we use an analytical quotient in place of the division operator to avoid division by zero errors. For two inputs, a and b , the analytical quotient is defined as $AQ(a, b) = \frac{b}{\sqrt{1+a^2}}$.

2.5.2 Large-Scale Experiments

In this section, we present the experiments conducted on 90 PMLB datasets and 40 DIGEN datasets to validate the effectiveness of the proposed method. Specifically, we compare our algorithm with traditional machine learning algorithms and popular deep learning-based methods. The distribution of balanced test accuracy and improvement of EvoFeat over other algorithms are presented in Figs. 2.3 and 2.4, respectively. A statistical comparison is shown in Table 2.3. The experimental results show that EvoFeat significantly outperforms state-of-the-art machine learning algorithms and deep learning methods. Moreover, the training time is presented in Fig. 2.5, showing

² <https://github.com/EpistasisLab/digen>.

³ <https://github.com/EpistasisLab/pmlb>.

Table 2.1 Hyperparameter spaces for various models

Model	Parameter	Distribution
XGBoost	Booster	['gbtree', 'dart']
	Number of estimators	UniformInt[10, 1000]
	Lambda	UniformLog[1e-5, 1e2]
	Alpha	UniformLog[1e-5, 1e2]
	Gamma	Uniform[0, 0.5]
	Learning rate	UniformLog[1e-8, 1]
	Max depth	UniformInt[1, 10]
RF	Number of estimators	UniformInt[10, 100]
	Criterion	['gini', 'entropy']
	Max depth	UniformInt[1, 10]
	Max features	[None, 'sqrt', 'log2']
	Min samples split	UniformInt[2, 20]
	Min samples leaf	UniformInt[1, 20]
LightGBM	Booster	['gbdt', 'dart', 'goss']
	Number of leaves	UniformInt[2, 256]
	Max depth	UniformInt[1, 10]
	Number of estimators	UniformInt[10, 1000]
DT	Criterion	['gini', 'entropy']
	Max depth	UniformInt[1, 10]
	Min samples split	UniformInt[2, 20]
	Min samples leaf	UniformInt[1, 20]
	Max features	[None, 'sqrt', 'log2']
KNN	Neighbors	UniformInt[1, 50]
	Weights	['uniform', 'distance']
	Power	UniformInt[1, 5]
	Metric	['euclidean', 'minkowski']
LR	Penalty	['l1', 'l2']
	Weights	UniformLog[1e-4, 1e4]
DNN	Layers	UniformInt[1, 8]
	Layer size	UniformInt[1, 512]
	Embedding size	UniformInt[64, 512]
	Dropout	Uniform[0, 0.5]
	Learning rate	UniformLog[1e-4, 1e-1]
	Weight decay	UniformLog[1e-6, 1e-3]
DCN V2	Cross layers	UniformInt[1, 8]
	Hidden layers	UniformInt[1, 8]
	Layer size	UniformInt[64, 512]
	Hidden dropout	Uniform[0, 0.5]
	Cross dropout	Uniform[0, 0.5]
	Embedding size	UniformInt[64, 512]
	Learning rate	UniformLog[1e-4, 1e-1]
	Weight decay	UniformLog[1e-6, 1e-3]

(continued)

Table 2.1 (continued)

Model	Parameter	Distribution
ResNet	Hidden layers	UniformInt[1, 8]
	Layer size	UniformInt[64, 512]
	Hidden factor	Uniform[1, 4]
	Hidden dropout	Uniform[0, 0.5]
	Residual dropout	Uniform[0, 0.5]
	Embedding size	UniformInt[64, 512]
	Learning rate	UniformLog[1e-4, 1e-1]
	Weight decay	UniformLog[1e-6, 1e-3]
FTTransformer	Attention dropout	Uniform[0,0.5]
	Residual dropout	Uniform[0, 0.2]
	FFN dropout	Uniform[0, 0.5]
	FFN factor	Uniform[$\frac{2}{3}$, $\frac{8}{3}$]
	Token dimension	UniformInt[64, 512]
	Layers	UniformInt[1, 4]
	Learning rate	UniformLog[1e-4, 1e-1]
	Weight decay	UniformLog[1e-6, 1e-3]

Table 2.2 Hyperparameter settings for EvoFeat

Parameter	Value
Population size	200
Maximal number of generations	30
Crossover and mutation rates	0.9 and 0.1
Maximum tree depth	8
Initial tree depth	0–2
Maximum forest size	100
Number of trees	20
Function set	+, −, *, Analytical Quotient, Square Root, Sin, Cos, Maximum, Minimum, Negative

that EvoFeat has a comparable training time to a fine-tuned LightGBM, indicating the high efficiency of the evolutionary feature construction method.

2.5.2.1 Comparison with Traditional Machine Learning Methods

Baseline Algorithms: First, we compare the proposed EvoFeat algorithm with traditional machine learning algorithms to demonstrate how feature construction enhances performance. We evaluate our algorithm against six traditional machine learning classification algorithms, including XGBoost [6], LightGBM [12], Random Forest (RF), Decision Tree (DT), Logistic Regression (LR), and K-Nearest Neighbors

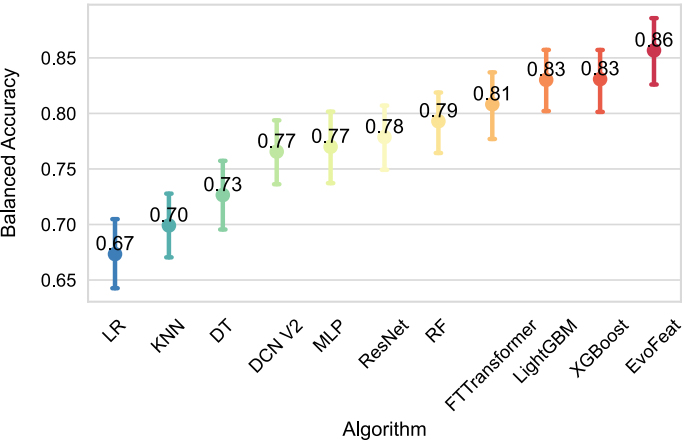


Fig. 2.3 Comparison of balanced test accuracy across different algorithms on 90 PMLB datasets and 40 DIGEN datasets (Dots represent mean values, and bars indicate 95% confidence intervals)

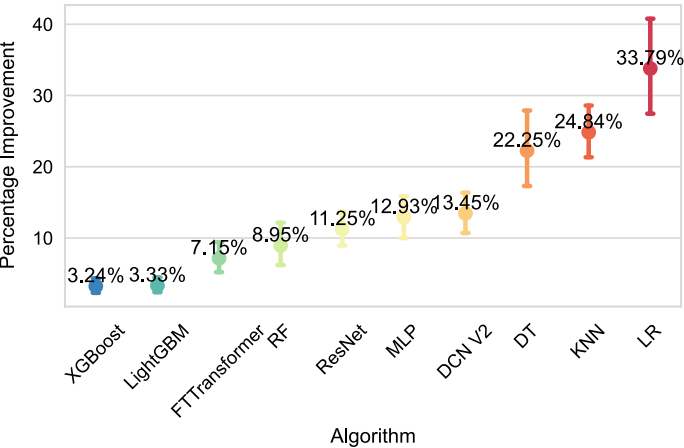


Fig. 2.4 Improvement in balanced test accuracy distributions across different algorithms on 90 PMLB datasets and 40 DIGEN datasets (Dots represent mean values, and bars indicate 95% confidence intervals)

(KNN). Among these, XGBoost and LightGBM are considered state-of-the-art algorithms [9].

Experimental Results: Figure 2.3 illustrates the test accuracy of all algorithms across 130 datasets. Gradient boosting methods such as XGBoost and LightGBM emerge as top-performing classification algorithms. However, our approach achieves better average accuracy than these algorithms. The numerical results presented in Table 2.3 confirm this, as our method achieves an average accuracy of 86%, which is a more than 3% improvement over the average accuracies of XGBoost and LightGBM.

Table 2.3 Statistical results of balanced test accuracy for different algorithms on 90 PMLB datasets and 40 DIGEN datasets (“+”, “~”, and “-” indicate that using the method in a row is better than, similar to, or worse than using the method in a column, respectively)

	XGBoost	LightGBM	RF	LR
DT	0(+)/48(~)/82(-)	2(+)/47(~)/81(-)	0(+)/43(~)/87(-)	60(+)/36(~)/34(-)
XGBoost	-	13(+)/107(~)/10(-)	43(+)/79(~)/8(-)	72(+)/50(~)/8(-)
LightGBM	-	-	45(+)/75(~)/10(-)	74(+)/42(~)/14(-)
RF	-	-	-	73(+)/47(~)/10(-)
	KNN	MLP	FTTransformer	DCN V2
DT	60(+)/27(~)/43(-)	36(+)/36(~)/58(-)	24(+)/47(~)/59(-)	37(+)/36(~)/57(-)
XGBoost	107(+)/16(~)/7(-)	73(+)/40(~)/17(-)	59(+)/56(~)/15(-)	77(+)/36(~)/17(-)
LightGBM	107(+)/15(~)/8(-)	75(+)/39(~)/16(-)	57(+)/56(~)/17(-)	79(+)/38(~)/13(-)
RF	102(+)/20(~)/8(-)	59(+)/40(~)/31(-)	51(+)/43(~)/36(-)	64(+)/37(~)/29(-)
LR	54(+)/13(~)/63(-)	30(+)/36(~)/64(-)	30(+)/40(~)/60(-)	37(+)/32(~)/61(-)
KNN	-	21(+)/23(~)/86(-)	17(+)/30(~)/83(-)	20(+)/29(~)/81(-)
MLP	-	-	10(+)/76(~)/44(-)	9(+)/118(~)/3(-)
FTTransformer	-	-	-	45(+)/79(~)/6(-)
	ResNet	EvoFeat		
DT	29(+)/48(~)/53(-)	0(+)/34(~)/96(-)		
XGBoost	76(+)/43(~)/11(-)	4(+)/67(~)/59(-)		
LightGBM	80(+)/39(~)/11(-)	5(+)/72(~)/53(-)		
RF	62(+)/44(~)/24(-)	7(+)/62(~)/61(-)		
LR	28(+)/39(~)/63(-)	7(+)/44(~)/79(-)		
KNN	15(+)/34(~)/81(-)	3(+)/15(~)/112(-)		
MLP	18(+)/96(~)/16(-)	4(+)/33(~)/93(-)		
FTTransformer	46(+)/73(~)/11(-)	4(+)/34(~)/92(-)		
DCN V2	8(+)/99(~)/23(-)	2(+)/35(~)/93(-)		
ResNet	-	3(+)/32(~)/95(-)		

Further analysis reveals that EvoFeat significantly outperforms XGBoost on 59 out of 130 datasets, while it is worse on only 4 datasets. These results demonstrate that our evolution-based feature construction method enables an ensemble learning model composed of simple base learners to outperform state-of-the-art machine learning models.

2.5.2.2 Comparison with Deep Learning Methods

Baseline Algorithms: In recent years, several deep learning-based methods have been proposed for tabular data learning. We have selected four models from the existing literature for comparison: multilayer perceptron (MLP), ResNet, DCN V2 [37], and FT-Transformer [9]. The first two models are commonly used baselines in the existing literature. The third model is the state-of-the-art model for click-through

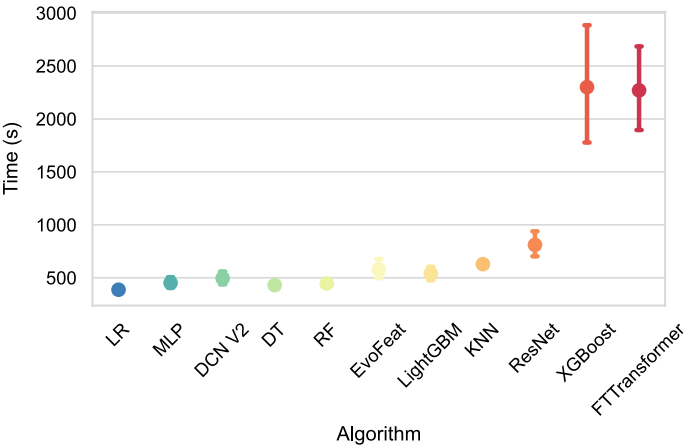


Fig. 2.5 Comparison of training time (in seconds) across different algorithms on 90 PMLB datasets and 40 DIGEN datasets (Dots represent mean values, and bars indicate 95% confidence intervals)

rate (CTR) prediction, and the fourth is the state-of-the-art model for tabular data learning.

Experimental Results: The experimental results in Fig. 2.3 reveal that existing deep learning approaches cannot compete with XGBoost or LightGBM, as the best-performing model, the FT-Transformer, achieves an average accuracy of only 81%. The state-of-the-art tabular learning methods exhibit performance only comparable to that of a fine-tuned random forest, which has an average accuracy of 79%. One probable explanation is that these deep learning algorithms require a large amount of training data and thus fail to learn robust feature representations in datasets with fewer than ten thousand items, which is common in tabular data.

2.5.3 Ablation Studies

In this paper, we make two improvements to the evolution-based feature construction method, i.e., constructs an ensemble of heterogeneous base learners and guides feature search using feature importance values on top of optimizing cross-validation losses. In this section, we present two sets of experiments to verify the validity of these two enhancements.

2.5.3.1 Base Learners

Due to the challenges in generating a diverse set of linear models, most ensemble learning systems employ decision trees as base learners rather than linear regression

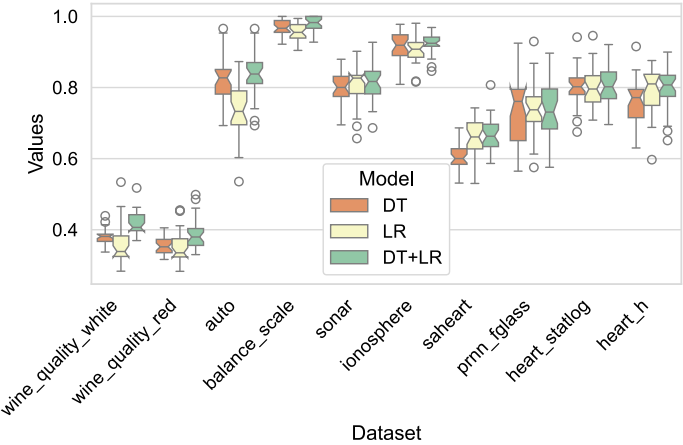


Fig. 2.6 Balanced test accuracy of our algorithm integrated with different base learners

Table 2.4 Comparison of balanced test accuracy across different base learners on 90 PMLB datasets

	LR	DT+LR
DT	12(+)/47(~)/31(-)	0(+)/62(~)/28(-)
LR	–	5(+)/70(~)/15(-)

models. However, constructing non-linear features may lead to different outcomes. Therefore, we conducted an ablation study in which we used only linear regression models or decision trees during the evolutionary feature construction process to investigate the benefits of simultaneously constructing features for both linear regression models and decision trees.

Figure 2.6 illustrates the balanced test accuracy with various base learners across 10 datasets in PMLB. The experimental results reveal that there is no free lunch when it comes to solving classification problems, and both decision trees and linear regression have their specific niche tasks. Nonetheless, the combination of random decision trees and logistic regression models enables our system to automatically select the most suitable model on the fly, resulting in improved average performance. Table 2.4 shows the statistical comparison of using different kinds of base learners across 90 PMLB datasets. This table demonstrates that the combination of DT and LR achieves significantly better performance compared to solely using LR on 15 datasets and only significantly worse results on 5 datasets. Comparing DT+LR with purely using DT, DT+LR achieves significantly better performance on 28 datasets, and no worse on any dataset. All these results confirm the effectiveness of employing heterogeneous base learners.

2.5.3.2 Feature Importance-Guided Evolutionary Search

In this paper, we propose feature importance-based genetic operators to exploit information provided by feature importance on top of cross-validation loss. It is worthwhile to investigate whether the feature importance-based genetic operators effectively use this information during the search process. Otherwise, random crossover and random mutation would be preferable options because they are much simpler. Therefore, this section presents an ablation experiment to demonstrate the efficacy of feature importance-guided genetic operators.

Four genetic operators, including random crossover and mutation (Random), random crossover with guided mutation (GM), guided mutation with self-competitive crossover (SS), and guided mutation with softmax selection-based self-competitive crossover (SC), are compared in this section. Figure 2.7 presents the distribution of test accuracy with respect to four genetic operators on 10 DIGEN datasets. From this figure, it is clear that the combination of guided mutation and softmax selection-based self-competitive crossover can effectively enhance the search process and ultimately result in better predictive performance.

Table 2.5 shows a statistical comparison of different combinations of crossover and mutation operators on 40 DIGEN datasets. These results confirm the effectiveness of incorporating softmax selection with self-competitive crossover operators to leverage feature importance information. Compared to random crossover and mutation operators, the SC operator with GM can outperform on 12 datasets and does not underperform on any datasets. The statistical comparison shown in Table 2.5 also indicates that the improvement comes from both GM and SC, implying that leveraging feature importance information in crossover is important. It is worth noting that in the crossover operator, it is important to use softmax to highlight important and less useful features compared to using naive self-competitive crossover, as the softmax

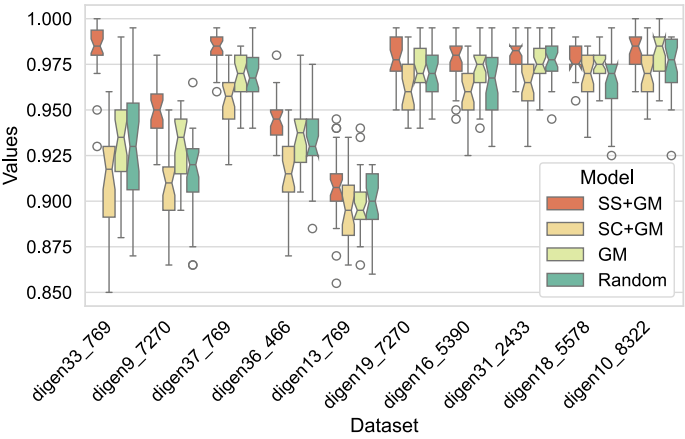


Fig. 2.7 Balanced test accuracy of our algorithm using different selection operators

Table 2.5 Comparison of balanced test accuracy across different selection operators

	SC+GM	GM	Random
SS+GM	12(+)/26(~)/2(-)	5(+)/34(~)/1(-)	12(+)/28(~)/0(-)
SC+GM	–	0(+)/30(~)/10(-)	5(+)/30(~)/5(-)
GM	–	–	5(+)/35(~)/0(-)

selection-based self-competitive crossover can outperform the naive version on 12 datasets and underperform on 2 datasets, indicating the effectiveness of the softmax selection. In conclusion, all these experimental results suggest that leveraging both cross-validation loss and feature importance information is beneficial for obtaining better final results.

2.6 Conclusions

Automated feature engineering is a critical component of an automated machine learning system. In this research, we have developed and presented a GP-based feature construction method to assist standard machine learning algorithms. To make the technique applicable to a wide range of tasks, we simultaneously evolve features for decision trees and linear regression models and form an ensemble of top-performing models. To increase the search effectiveness of our algorithm, we designed a softmax selection operator that uses feature importance values to utilize promising features. Experimental results on 130 datasets show that the proposed method outperforms both state-of-the-art tree-based ensemble learning methods and deep learning methods. Furthermore, the ablation studies show that letting the evolutionary algorithm automatically choose base learners for classification is beneficial and that the proposed softmax selection operator is effective in guiding the search process.

For future study, considering that the features useful for decision trees and logistic regression models may share some common sub-expressions, it is worth investigating whether a modularization technique [41] could be used to improve the interpretability of the ensemble system. Furthermore, explicitly considering the semantic complementarity [40] might be another promising direction to further improve the ensemble performance.

References

1. Allen-Zhu, Z., Li, Y.: What can resnet learn efficiently, going beyond kernels? Adv. Neural Inform.n Process. Syst. **32**, 9015–9025 (2019)
2. Arik, S.O., Pfister, T.: Tabnet: attentive interpretable tabular learning. In: AAAI, vol. 35, pp. 6679–6687 (2021)

3. Burke, E., Gustafson, S., Kendall, G.: Ramped half-n-half initialisation bias in GP. In: Genetic and Evolutionary Computation Conference, pp. 1800–1801. Springer (2003)
4. Caruana, R., Niculescu-Mizil, A., Crew, G., Ksikes, A.: Ensemble selection from libraries of models. In: Proceedings of the Twenty-First International Conference on Machine Learning, p. 18 (2004)
5. Chao, S.K., Cheng, G.: A generalization of regularized dual averaging and its dynamics (2019). arXiv preprint [arXiv:1909.10072](https://arxiv.org/abs/1909.10072)
6. Chen, T., Guestrin, C.: XGBoost: A scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 785–794. ACM (2016)
7. Colange, B., Peltonen, J., Aupetit, M., Dutykh, D., Lespinats, S.: Steering distortions to preserve classes and neighbors in supervised dimensionality reduction. *Adv. Neural Inform. Process. Syst.* **33**, 13214–13225 (2020)
8. Cowen-Rivers, A.I., Lyu, W., Tutunov, R., Wang, Z., Grosnit, A., Griffiths, R.R., Maraval, A.M., Jianye, H., Wang, J., Peters, J., et al.: An empirical study of assumptions in Bayesian optimisation (2020). arXiv preprint [arXiv:2012.03826](https://arxiv.org/abs/2012.03826)
9. Gorishniy, Y., Rubachev, I., Khrulkov, V., Babenko, A.: Revisiting deep learning models for tabular data. *Adv. Neural Inform. Process. Syst.* **34**, 18932–18943 (2021)
10. Guo, H., Tang, R., Ye, Y., Li, Z., He, X.: DeepFM: a factorization-machine based neural network for CTR prediction. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence, pp. 1725–1731 (2017)
11. Katz, G., Shin, E.C.R., Song, D.: Exploreskit: automatic feature generation and selection. In: 2016 IEEE 16th International Conference on Data Mining (ICDM), pp. 979–984. IEEE (2016)
12. Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., Liu, T.Y.: LightGBM: a highly efficient gradient boosting decision tree. *Adv. Neural Inform. Process. Syst.* **30**, 3146–3154 (2017)
13. Kidger, P., Lyons, T.: Universal approximation with deep narrow networks. In: Conference on Learning Theory, pp. 2306–2327. PMLR (2020)
14. Kotanchek, M., Smits, G., Vladislavleva, E.: Pursuing the pareto paradigm: tournaments, algorithm variations and ordinal optimization. In: Genetic Programming Theory and Practice IV, pp. 167–185 (2007)
15. Koza, J.R.: Genetic programming as a means for programming computers by natural selection. *Stat. Comput.* **4**(2), 87–112 (1994)
16. La Cava, W., Helmuth, T., Spector, L., Moore, J.H.: A probabilistic and multi-objective analysis of lexibase selection and ϵ -lexibase selection. *Evol. Comput.* **27**(3), 377–402 (2019)
17. La Cava, W., Silva, S., Danai, K., Spector, L., Vanneschi, L., Moore, J.H.: Multidimensional genetic programming for multiclass classification. *Swarm Evol. Comput.* **44**, 260–272 (2019)
18. Lensen, A., Xue, B., Zhang, M.: Genetic programming for manifold learning: preserving local topology. *IEEE Trans. Evol. Comput.* (2021)
19. Lian, J., Zhou, X., Zhang, F., Chen, Z., Xie, X., Sun, G.: xDeepFM: Combining explicit and implicit feature interactions for recommender systems. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 1754–1763 (2018)
20. Liu, B., Zhu, C., Li, G., Zhang, W., Lai, J., Tang, R., He, X., Li, Z., Yu, Y.: Autofis: automatic feature interaction selection in factorization models for click-through rate prediction. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 2636–2645 (2020)
21. Liu, H., Simonyan, K., Yang, Y.: Darts: Differentiable architecture search. In: International Conference on Learning Representations (2018)
22. Lundberg, S.M., Lee, S.I.: A unified approach to interpreting model predictions. *Adv. Neural Inform. Process. Syst.* **30**, 4765–4774 (2017)
23. Luo, Y., Wang, M., Zhou, H., Yao, Q., Tu, W.W., Chen, Y., Dai, W., Yang, Q.: Autocross: automatic feature crossing for tabular data in real-world applications. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 1936–1945 (2019)

24. Meng, Z., Zhang, J., Li, Y., Li, J., Zhu, T., Sun, L.: A general method for automatic discovery of powerful interactions in click-through rate prediction. In: Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 1298–1307 (2021)
25. Mudgal, S., Li, H., Rekatsinas, T., Doan, A., Park, Y., Krishnan, G., Deep, R., Arcaute, E., Raghavendra, V.: Deep learning for entity matching: a design space exploration. In: Proceedings of the 2018 International Conference on Management of Data, pp. 19–34 (2018)
26. Muñoz, L., Silva, S., Trujillo, L.: M3gp—multiclass classification with GP. In: European Conference on Genetic Programming, pp. 78–91. Springer (2015)
27. Nag, K., Pal, N.R.: Feature extraction and selection for parsimonious classifiers with multiobjective genetic programming. *IEEE Trans. Evol. Comput.* **24**(3), 454–466 (2019)
28. Nguyen, S., Thiruvady, D., Zhang, M., Alahakoon, D.: Automated design of multipass heuristics for resource-constrained job scheduling with self-competitive genetic programming. *IEEE Trans. Cybern.* 1–14 (2021)
29. Orzechowski, P., Moore, J.H.: Generative and reproducible benchmarks for comprehensive evaluation of machine learning classifiers (2021). arXiv preprint [arXiv:2107.06475](https://arxiv.org/abs/2107.06475)
30. Romano, J.D., Le, T.T., La Cava, W., Gregg, J.T., Goldberg, D.J., Chakraborty, P., Ray, N.L., Himmelstein, D., Fu, W., Moore, J.H.: Pmlb v1.0: an open source dataset collection for benchmarking machine learning methods (2021). arXiv preprint [arXiv:2012.00058v2](https://arxiv.org/abs/2012.00058v2)
31. Schmidt, M., Lipson, H.: Age-fitness pareto optimization. In: Genetic Programming Theory and Practice VIII, pp. 129–146. Springer (2011)
32. Shi, Q., Zhang, Y.L., Li, L., Yang, X., Li, M., Zhou, J.: Safe: scalable automatic feature engineering framework for industrial tasks. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE), pp. 1645–1656. IEEE (2020)
33. Song, W., Shi, C., Xiao, Z., Duan, Z., Xu, Y., Zhang, M., Tang, J.: Autoint: automatic feature interaction learning via self-attentive neural networks. In: Proceedings of the 28th ACM International Conference on Information and Knowledge Management, pp. 1161–1170 (2019)
34. Tran, B., Xue, B., Zhang, M.: Genetic programming for multiple-feature construction on high-dimensional classification. *Pattern Recogn.* **93**, 404–417 (2019)
35. Vanschoren, J., Van Rijn, J.N., Bischl, B., Torgo, L.: Openml: networked science in machine learning. *ACM SIGKDD Explor. Newsl.* **15**(2), 49–60 (2014)
36. Wang, C., Chen, Q., Xue, B., Zhang, M.: Shapley value based feature selection to improve generalization of genetic programming for high-dimensional symbolic regression. In: Australasian Conference on Data Science and Machine Learning, pp. 163–176. Springer (2023)
37. Wang, R., Shivanna, R., Cheng, D., Jain, S., Lin, D., Hong, L., Chi, E.: DCN V2: improved deep & cross network and practical lessons for web-scale learning to rank systems. In: Proceedings of the Web Conference, pp. 1785–1797 (2021)
38. Wei, T., Wang, S., Zhong, J., Liu, D., Zhang, J.: A review on evolutionary multitask optimization: trends and challenges. *IEEE Trans. Evol. Comput.* **26**(5), 941–960 (2021)
39. Yang, H., Wang, T., Tang, X., Li, Q., Shi, Y., Jiang, S., Yu, H., Song, H.: Multi-task learning for bias-free joint CTR prediction and market price modeling in online advertising. In: Proceedings of the 30th ACM International Conference on Information & Knowledge Management, pp. 2291–2300 (2021)
40. Zhang, H., Chen, Q., Xue, B., Banzhaf, W., Zhang, M.: MAP-Elites for genetic programming-based ensemble learning: an interactive approach [AI-explained]. *IEEE Comput. Intel. Mag.* **18**(4), 62–63 (2023)
41. Zhang, H., Chen, Q., Xue, B., Banzhaf, W., Zhang, M.: Modular multi-tree genetic programming for evolutionary feature construction for regression. *IEEE Trans. Evol. Comput.* (2023). <https://doi.org/10.1109/TEVC.2023.3318638>
42. Zhang, H., Chen, Q., Xue, B., Banzhaf, W., Zhang, M.: A semantic-based hoist mutation operator for evolutionary feature construction in regression. *IEEE Trans. Evol. Comput.* (2023). <https://doi.org/10.1109/TEVC.2023.3331234>
43. Zhang, H., Zhou, A., Chen, Q., Xue, B., Zhang, M.: SR-Forest: a genetic programming based heterogeneous ensemble learning method. *IEEE Trans. Evol. Comput.* (2023). <https://doi.org/10.1109/TEVC.2023.3243172>

44. Zhang, H., Zhou, A., Zhang, H.: An evolutionary forest for regression. *IEEE Trans. Evol. Comput.* 1–1 (2021). <https://doi.org/10.1109/TEVC.2021.3136667>
45. Zhang, Q., Sun, J., Tsang, E.: An evolutionary algorithm with guided mutation for the maximum clique problem. *IEEE Trans. Evol. Comput.* **9**(2), 192–200 (2005)

Chapter 3

Deep Learning-Based Operators for Evolutionary Algorithms



Eliad Shem-Tov, Moshe Sipper, and Achiya Elyasaf

Abstract We present two novel *domain-independent* genetic operators that harness the capabilities of deep learning: a crossover operator for genetic algorithms and a mutation operator for genetic programming. Deep Neural Crossover leverages the capabilities of deep reinforcement learning and an encoder-decoder architecture to select offspring genes. BERT mutation masks multiple gp-tree nodes and then tries to replace these masks with nodes that will most likely improve the individual's fitness. We show the efficacy of both operators through experimentation.

3.1 Introduction

Two major components in a genetic algorithm (GA) are the crossover and mutation operators. These operators traditionally rely on a random selection of parental genes and random mutations. Throughout the years, numerous operators have been proposed to improve the solution quality obtained by the population. Often, these operators are tailored to specific problem domains, such as the traveling salesman problem (TSP) [1, 2], feature selection [17], job scheduling [18], and more.

This paper presents two novel *domain-independent* operators that harness the capabilities of deep learning techniques: a crossover operator for genetic algorithms and a mutation operator for genetic programming. Our aim is to create operators that can seamlessly adapt to different problem domains, without requiring hand-crafted adjustments to each problem domain.

E. Shem-Tov · A. Elyasaf

Department of Software and Information Systems Engineering, Ben-Gurion University of the Negev, Beer-Sheva 8410501, Israel
e-mail: eliads@post.bgu.ac.il

A. Elyasaf

e-mail: achiya@bgu.ac.il

M. Sipper (✉)

Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva 8410501, Israel
e-mail: sipper@bgu.ac.il

Using deep learning algorithms for crossover and mutation operators may raise concerns about increased time and computational resources compared to standard crossover methods such as uniform crossover. It is important to note that the training of our operators does not necessitate additional fitness evaluations. Instead, they use online learning, where the fitness values calculated during the evolutionary process are also employed for training the operators.

That said, to address the time-related concerns, we propose a transfer-learning approach: Initially, we train the architecture on a single problem within a specific domain (e.g., a single bin-packing instance), and then we apply the trained operator to solve other problems within the same domain.

3.2 Deep Neural Crossover

We presented a novel, multi-parent crossover operator for genetic algorithms (GAs) called *Deep Neural Crossover* (DNC) in [24]. Unlike conventional GA crossover operators that rely on a random selection of parental genes, DNC leverages the capabilities of deep reinforcement learning (DRL) and an encoder-decoder architecture to select the genes.

Specifically, we used DRL to learn a policy for selecting promising genes. The policy is stochastic, to maintain the stochastic nature of GAs, representing a distribution for selecting genes with a higher probability of improving fitness. Our architecture features a recurrent neural network (RNN) to encode the parental genomes into latent memory states, and a decoder RNN that uses an attention-based pointing mechanism to generate a distribution over the next selected gene in the offspring.

Crossover operators, such as *one-point crossover* and *uniform crossover*, rely on a random selection of parental genes. There is a key distinction between these two: The former assumes a certain order between genes, suggesting that closer genes are more likely to be passed on together. In contrast, uniform crossover does away with this assumption, assuming no correlation. Our approach represents a progression from this line of thought. The DNC operator is the first crossover operator designed to learn correlations between genes, eschewing pre-existing, possibly biased assumptions. These correlations may either be linear or nonlinear.

To reduce the computational cost we further presented a transfer-learning approach, wherein the architecture is initially trained on a single problem within a specific domain—and then applied to solving other problems within the same domain.

We compared DNC to known operators from the literature over two benchmark domains, outperforming all baselines.

3.2.1 Previous Work

Arguably, the most popular crossover operators in the literature are “one point” and “uniform” crossover operators [9]. With one point, a single crossover point is selected randomly, while with uniform, each gene position is independently selected with equal probability from either parent. When uniform crossover uses more than two parents, the operator is called multi-parent uniform crossover.

In this paper, we refer to the uniform crossover as *equiprobable uniform* to differentiate it from the adaptive uniform operator, as we now elaborate.

Semenkin and Semenkina [23] proposed an adaptation to the equiprobable uniform crossover. In their modification, the probability of inheriting a gene from a parent is influenced by the ratio of fitness values between the two parents. This approach aims to dynamically alter the gene selection process, considering the relative fitness contributions of each parent during the evolution process.

Kiraz et al. merged the concepts of multi-parent and adaptive uniform crossover with collective crossover [14]. They suggested constructing an individual by incorporating the entire population, diverging from conventional uniform crossover methods where only two individuals are involved, or standard multi-parent crossovers where a fixed number of parents are employed. The collective operator aims to enhance the exploration capability of GAs by leveraging information from the entire population during the crossover process. The probability of selecting a gene from each individual within the population dynamically changes based on their fitness performance. Multi-parent crossover and the adaptive operator of Semenkin and Semenkina [23] are part of our baseline operators that we compare with.

We found only one work in the literature that uses machine learning for a domain-independent crossover. Liu et al. [16] proposed NeuroCrossover, an operator that leverages a transformer model coupled with a reinforcement learning (RL) approach to optimize the selection of multiple crossover points. They applied their NeuroCrossover model to determine the location of crossover points, exemplified through its application to a 2-point crossover scenario.

Our approach in Sect. 3.2.2 is similar to this approach in that both use RL. Nevertheless, there are some significant differences: Our approach allows for running multi-parent crossover, and as we demonstrate below, it dramatically improves the results. In addition, NeuroCrossover optimizes the selection of two crossover points and, thus, detects only order-related correlations, like n -point crossover. DNC, on the other hand, selects each gene sequentially, one by one, allowing us to also use nonlinear gene correlations. We compare our operator with NeuroCrossover in Sect. 3.2.4.

3.2.2 Method

Our operator produces a single child by merging the genes of the parents. It can generate exactly the same offspring as the uniform crossover, with a difference in the distribution for generating each offspring. That is, DNC tries to assign a higher probability to offspring with a high fitness score, in contrast to uniform crossover, where all offspring have an equal probability of being generated.

To learn these probabilities and generate the offspring, we use a sequence-to-sequence architecture, presented by [4], which consists of two recurrent neural network (RNN) modules: an encoder and a decoder. The encoder network (Fig. 3.1) maps the two parents into a single embedded representation, and the decoder (Fig. 3.2) learns to generate a new child from this representation.

To represent the distribution for selecting a child gene, we use a *pointer network* [27]. This network uses a set of softmax modules with attention to produce

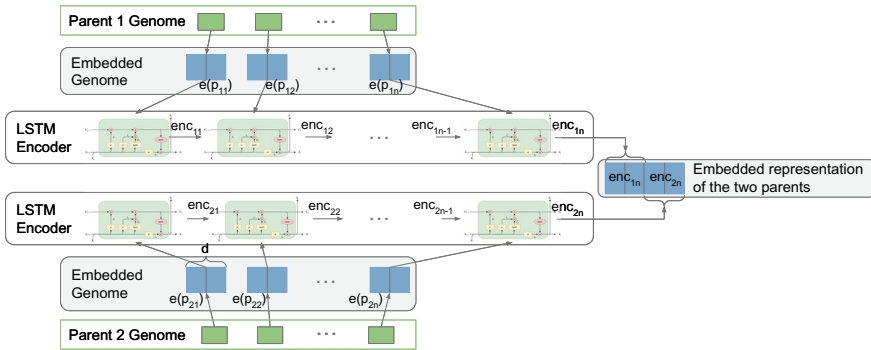


Fig. 3.1 Encoder architecture. Parents' genes are embedded into a shared latent feature space and subsequently processed by an LSTM network to produce an embedded representation of the parents

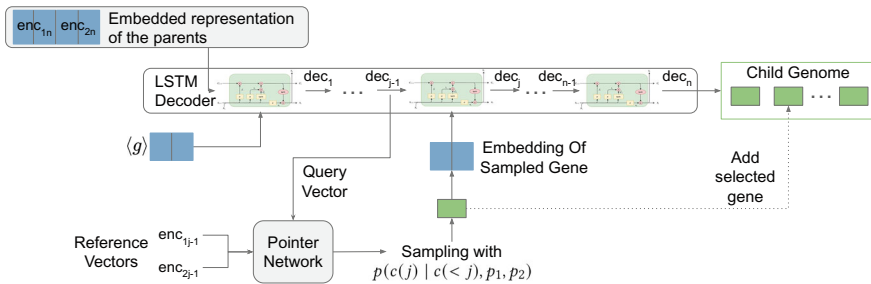


Fig. 3.2 Decoder architecture. The decoder input is the encoder's output, i.e., the embedded representation of the two parents, which is subsequently processed and decoded by an LSTM network to produce a child genome. At each decoding step, the output is sent to a pointer network that chooses a gene from one of the parents and returns an embedded representation of the sampled gene. Thus, the child's genome is constructed gene by gene, from left to right, starting from an empty genome

the probability of selecting a gene from each parent. It is called a pointer network because it uses a combination of attention and softmax modules to effectively point to a specific position in the input sequence.

We use policy-based reinforcement learning to train both the LSTM and the pointer network, as there is no ground truth for training the architecture. Since our training objective is to maximize the fitness score, we defined the reward signal as the fitness function of the generated offspring. We aimed to improve the expected fitness score of the generated offspring that is sampled from the policy distribution. We optimized our network using the policy gradient method REINFORCE [28], with stochastic gradient descent.

Training does not necessitate additional fitness evaluations since the fitness values calculated during the evolutionary process are used for training the operator.

3.2.3 Enhancements

Pointer networks are not limited to two reference vectors. Thus, our DNC operator supports multi-parent crossover. Specifically, when using $m \geq 2$ parents, the encoder's output is an embedded representation of all parents, and the reference vectors for the pointer network are the corresponding hidden encoder states of all parents. In Sect. 3.2.4, we compare two- and three-parent versions of our operator to the baseline operators.

The training process of our DNC operator consumes more computation than standard crossover (e.g., uniform crossover). As demonstrated below, these extra resources are worthwhile, given the much-improved results.

To improve computation time, we propose a pre-training approach, wherein the architecture is initially trained on a single problem within a specific domain—and then applied to solving other problems of the same domain.

3.2.4 Evaluation

To assess the performance of the proposed operator, we carried out an extensive set of experiments on two problem domains—Graph Coloring and Bin Packing (BPP)—using the DIMACS [12] and Schoenfeld_Hard28 [22] datasets as benchmarks for these domains (respectively).

We compared our approach to the following domain-independent crossover operators: Single Point [9], Equiprobable Uniform [9] (aka uniform crossover), Adaptive Uniform [23] (a variant of the equiprobable uniform, where the probability of inheriting a gene from a particular parent is dynamically adjusted based on the ratio of parents fitness), Multi-Parent Equiprobable Uniform [8], and NeuroCrossover [16] (a 2-point crossover operator that employs RL and an encoder-decoder transformer architecture to optimize the genetic locus selection).

DNC significantly outperformed all baselines by a wide margin, finding near-optimal solutions in both domains. The pre-training approach is trained on a single instance in each domain and then used the trained operator to evolve all other instances. Resultant performance is comparable to the basic DNC operator and, in most cases, better than the baseline operators. These results were obtained with a notable reduction in runtime compared to the base DNC algorithm. The complete experimental details and results can be found in [24].

Figure 3.3 shows the impact of different crossover operators on the maximum fitness value per generation. Each plot line represents a different crossover operator, allowing for a direct comparison of their performance over successive generations. Multi-Parent DNC outperforms all other crossover operators, with a clear improvement in solution quality and convergence speed. This trend persists across other problem instances.

The figure also visualizes the trade-off between the runtime and the achieved solution quality. The cutoff times are denoted in the graphs in Fig. 3.3. It is noteworthy that after 5 min, the DNC and DNC-MP are far better than the baselines, despite being in earlier generations.

We also analyzed the trade-off between runtime and solution quality. Our approach demonstrated a substantial improvement over the best baseline operator. Nevertheless, to address the problem of extra resources, we included time comparisons for our pre-trained suggested approach (DNC-PT), where the architecture is pre-trained on a specific instance within a problem domain and then applied to other problems. This

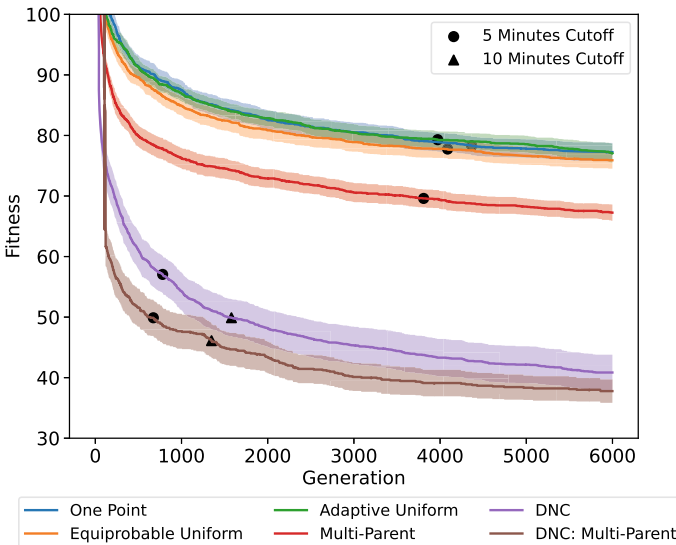


Fig. 3.3 Fitness value of best individual versus generation, of each operator for the `zero.in.i.2` graph-coloring problem. The fitness value is averaged over 20 runs. The black dots and triangles along each line mark the 5 and 10 min runtime cutoffs, respectively

results in a clear decrease in runtime, down to approximately 0.5 extra seconds per generation on average. Thus, the added cost is probably worth it given the far-better solutions obtained.

3.3 BERT Mutation

In conventional genetic programming (GP), the mutation operator may assume a dominant role, sometimes used as the sole operator, with no crossover whatsoever. A widely used mutation operator for GP is point mutation, wherein a random tree node is replaced with an appropriate random node, i.e., a random function with the same arity or a random terminal, depending on the replaced node type [21].

Our BERT mutation takes point mutation a step further. It masks multiple tree nodes and then tries to replace these masks with tree nodes that will most likely improve the individual’s fitness. The operator draws inspiration from natural language processing (NLP) techniques, particularly the Masked Language Modeling (MLM) approach used to train models like BERT (Bidirectional Encoder Representations from Transformers) [7].

MLM involves masking a certain percentage of tokens in input sentences and tasking the model to predict these masked tokens based on contextual information. MLM enables the model to learn rich contextual representations of language by optimizing the model to predict masked words. This approach revolutionized NLP, offering state-of-the-art performance across various downstream tasks and setting a new benchmark for contextualized embeddings (Fig. 3.4).

GP offers an intriguing case for integrating the MLM approach. Each individual within the population is a string drawn from a predefined language containing functions and terminals. Consider the GP tree depicted in Fig. 3.5, which can be represented as a string: $(2.2 - (x/11)) + (7 * \cos(y))$.

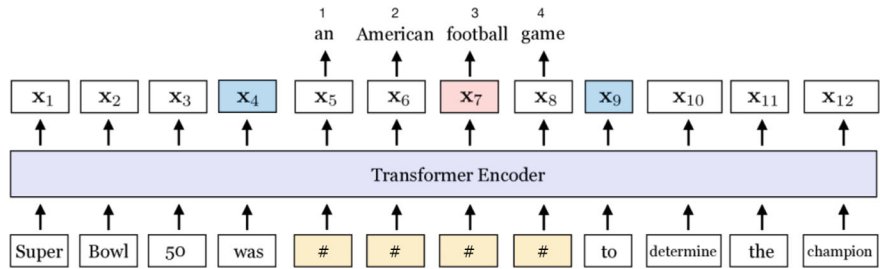
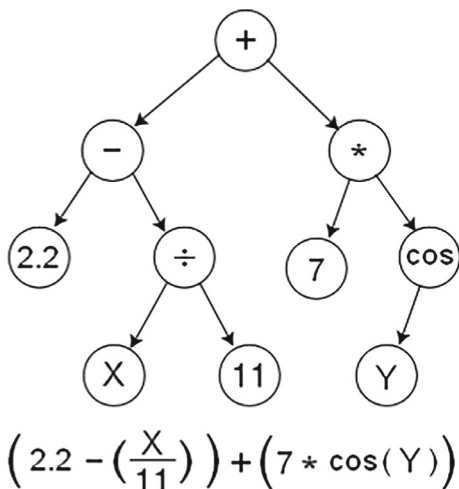


Fig. 3.4 Illustration of Masked Language Modeling using BERT [13]: “Super Bowl 50 was an American football game to determine the champion” becomes “Super Bowl 50 was # # # # to determine the champion,” where # represents a mask. The model is then trained to predict the masked tokens, thereby inferring the missing words. This approach enables the model to learn bidirectional contextual representations by incorporating both left and right contexts during training

Fig. 3.5 A GP tree

For example, we can obtain the following string from the masking procedure: $(\# - (x/\#)) + (7\# \cos(y))$. Inspired by the training process of BERT, we propose applying MLM to the string representation of GP trees along with the mutation masks. However, instead of optimizing the network to predict masked nodes, we use reinforcement learning to select new nodes that optimize the population’s fitness scores, where the fitness improvement serves as the reward signal.

In other words, we model a probability distribution for each masked node, aiming to assign higher probabilities to functions and terminals that are expected to enhance the fitness of the mutated individual. This optimization process ensures that the mutations introduced by evolution are more likely to lead to improved solutions.

3.3.1 Previous Work

Uy et al. [25] introduced semantic-based mutation operators for GP trees. Departing from random mutations, semantic mutation introduces alterations intended to modify behavior in predefined, controlled manners. These modifications are guided by desired changes in the program’s semantics. Semantic mutation strives for smooth and incremental changes rather than abrupt ones, mitigating sudden disruptions in program behavior. In their work, Uy et al. [25] proposed a variant of subtree mutation: Before substituting the new subtree, they verify that its semantic dissimilarity to the previous subtree is within a predefined acceptable range.

Blanchard et al. [5] suggested using similar concepts inspired by masked language models used for text generation. Even though this work is domain-specific and is applied only to molecule generation, we include it due to its similar concepts and ideas applied in our approach. The authors proposed a method that uses

mask prediction to create new molecule sequences during the mutation process. This approach eliminates the need for hand-crafted mutation rules and allows for incorporating larger subsequence rearrangements beyond single-point changes. In the proposed approach, a masked language model is trained on tokenized data of molecules to produce possible mutations for the evolutionary algorithm. As opposed to their suggested approach, our approach does not require a large dataset of individual strings in order to produce mutations, but rather learns online during evolution via reinforcement learning what mutations might lead to improved fitness scores.

A recent study by OpenAI [15] investigated the potential synergy between large language models (LLM) and evolutionary computation, particularly focusing on the Evolution through Large Models (ELM) approach. ELM leverages large language models trained on code to suggest intelligent mutations, enhancing the effectiveness of mutation operators in GP. They employed a *diff* model, which is an LLM that is optimized to predict the differences between code files based on their commit messages from GitHub data. The mutation process involves maintaining a set of commit messages, which are then randomly selected during evolution, and the model’s predicted diff is applied as a mutation to the code. In this way they make small tweaks to specific functions and segments of the code by choosing the appropriate commit message.

3.3.2 Method

The idea of our BERT mutation operator is simple in nature: we provide the BERT model with a masked version of a GP tree and “ask” it to predict the masked nodes in such a way that a possibly better individual is formed. This requires a modification of the training process. Since we do not have a ground-truth solution of the masking, we use reinforcement learning to train the model. We defined the reward signal as the total fitness improvement achieved over the pre-mutated individual. We use policy-based reinforcement learning to optimize the mutation process using the policy gradient method REINFORCE [28] with stochastic gradient descent.

Training the BERT model requires a dataset of individuals and their fitness. To train it without affecting the overall evolutionary speed, we cache the individuals and their fitness values during the evolutionary process. Once the cache reaches a specified batch size, we use it to train the BERT model. As we demonstrate in the results, the overall evolutionary speed is hardly affected.

During the evolutionary process, the operator randomly masks the individual’s string representation with a masking probability constant and then uses the BERT model to replace the masks, one at a time. We want each replacement to take into account the previous replacements. Thus, the replacement order is important. We chose to replace the masks in a depth-first search (DFS) manner to be aligned with the tree structure, i.e., we start with the parent, then go to the left child, and then the right child (recursively).

The replacement process for a given mask is depicted in Fig. 3.6. The trained BERT model produces a distribution over the possible replacements for the given mask. Next, the softmax activation of BERT chooses a replacement by sampling the distribution. The sampling is constrained to include only valid options. For instance, when replacing terminals, only terminal tokens are considered, and similarly, when replacing functions, only functions with the corresponding arity are allowed. This method ensures that replacements are contextually appropriate. Since the proposed mutation operator does not expand the size of the existing GP trees in the population, we introduce more variability to the population by using hoist and subtree mutations with a small probability.

We tokenize each GP-tree string representation by assigning an integer to all operators and terminals. Constant terminals are represented using the same tokenized integer representation. Additionally, when the model replaces a masked node with a constant, we draw a random float between predefined boundaries.

3.3.3 Evaluation

We used gplearn [11] and implemented our architecture using PyTorch [20] and the Transformers [29] python package.

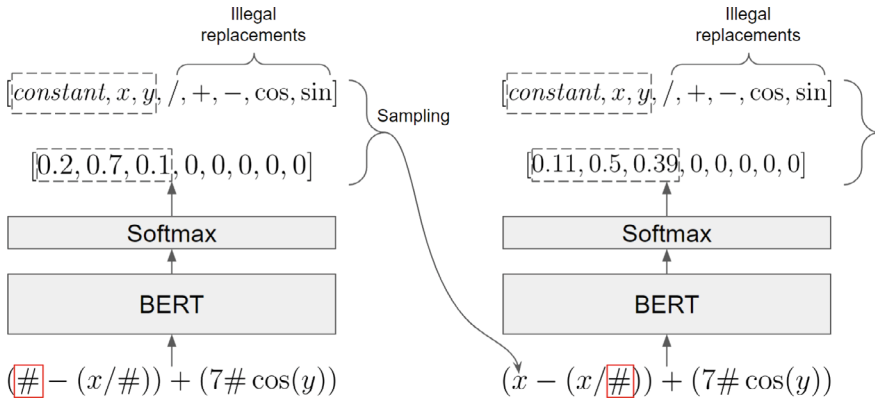


Fig. 3.6 The mask replacement process of the BERT mutation. The trained BERT model takes a masked individual and outputs the probability for each possible replacement. The softmax function samples a possible replacement, and the string with the replaced token is passed again until all masks are replaced

Datasets

To assess the performance of the proposed operator, we carried out extensive experiments on various datasets and equations for symbolic regression:

1. Airfoil Self-Noise regression dataset [6], containing 5 features and 1503 instances.
2. Concrete Compressive Strength regression dataset [30], containing 8 features and 1030 instances.
3. The Friedman 1–3 regression problems [10]. We generated 5000 instances using scikit-learn. The problems are defined as following:
 - a. Friedman-1 : $y = 10 \cdot \sin(\pi \cdot x_0 \cdot x_1) + 20(x_2 - 0.5)^2 + 10x_3 + 5x_4 + \text{noise} \cdot N(0, 1)$
 - b. Friedman-2 : $y = (x_0^2 + (x_1 \cdot x_2 - (\frac{1}{x_2 \cdot x_4}))^2)^{0.5} + \text{noise} \cdot N(0, 1)$
 - c. Friedman-3 : $y = \arctan(\frac{(x_1 \cdot x_2 - (\frac{1}{x_1 \cdot x_3}))}{x_0}) + \text{noise} \cdot N(0, 1)$
4. We generated 5000 instances between the ranges of $[-10, 10]$ for the following equations:
 - a. $f2 : y = (x_0 - 3)^4 + (x_1 - 2)^3$
 - b. non-analytic : $y = (x + 1)^2 \text{ if } x > 0 \text{ else } \sin(x)$.

Baselines

We compared against the following baseline mutation operators:

Hoist mutation. Hoist mutation [3] is a bloat-preventing mutation operation. The purpose of this mutation is to remove genetic material from individuals. In hoist mutation an individual from the population is selected and a random subtree within this individual is identified. From this selected subtree, a further random subtree is chosen. This second subtree, which is a subset of the first, is then “hoisted”, or elevated, to replace the original subtree of which it was a part. The result is that the individual’s overall structure is simplified, typically resulting in a more compact and potentially more efficient solution.

Subtree mutation. Subtree mutation [26] involves taking an individual and altering it by focusing on a specific segment of its structure. In this process, a random subtree within an individual is chosen to be replaced. To do this, a new subtree—often referred to as a donor subtree—is generated randomly. This donor subtree is then inserted into the original tree at the point of the removed subtree. The resulting modified tree, which integrates the new subtree, becomes an offspring in the next generation.

Point mutation. Point mutation [19] is used to modify an individual by selecting random nodes within it for replacement. In this process, terminals are substituted with other terminals, and functions are replaced with other functions that require the

Table 3.1 Comparison of mutation operators across various datasets for test-set fitness. The reported numbers represent the fitness of the best individual in the last generation. The numbers are averaged over 10 experiments. Lower values indicate better performance. Standard deviations are given in parentheses. Best results are boldfaced

Dataset	BERT mutation	Mixed mutation	Subtree mutation	Point mutation	Hoist mutation
AirfoilSelfNoise	7.76 (1.18)	13.20 (8.56)	14.38 (8.77)	20.09 (13.10)	27.84 (13.59)
concrete_data	11.42 (1.19)	12.34 (3.30)	12.68 (2.27)	15.13 (4.67)	14.87 (4.40)
friedman1	2.25 (0.12)	3.19 (1.14)	3.09 (0.94)	3.08 (1.04)	4.23 (0.77)
friedman2	7.23 (1.38)	16.79 (12.58)	11.14 (4.45)	17.63 (10.98)	73.38 (122.92)
friedman3	0.32 (0.03)	0.39 (0.03)	0.39 (0.03)	0.40 (0.01)	0.39 (0.04)
f2	5.68 (1.84)	11.01 (5.95)	7.97 (2.69)	14.88 (6.76)	18.37 (6.84)
non_analytic	0.34 (0.19)	2.94 (3.57)	2.22 (1.92)	2.65 (3.55)	5.67 (4.36)

same number of arguments as the original node. This selective alteration ensures that the structural integrity and functionality of the tree are maintained, while introducing genetic diversity. The modified tree, now sporting these changes, becomes an offspring in the subsequent generation. This mechanism is crucial for exploring the solution space and enhancing the genetic diversity within the population, potentially leading to improved solutions over successive generations.

Mixed mutation. We define mixed mutation as the mutation operator that uses all the previous three operators with equal probability for each operator to be chosen.

3.3.4 Results

We used a population size of 128 individuals run for 200 generations. We initialized the population using the ramped half-and-half technique, where the depth of grown trees was in the range of [2, 10]. We used a crossover probability of 0.6 with a subtree crossover probability and a mutation probability of 0.1 for BERT mutation and 0.05 for subtree and hoist mutation. The fitness measure used across all experiments was RMSE. We repeated each experiment 10 times and report on the average best fitness value of the last generation. We perform a train-test split with 10% of the data left to the test set.

The complete experimental results on the test set can be seen in Table 3.1. Notably, our BERT operator outperformed all other operators in each of the datasets.

Figure 3.7 shows the impact of the different mutation operators on the maximum fitness value per generation for a single instance of the non_analytic domain. Each plot line represents a different mutation operator, allowing for a direct comparison of their performance over successive generations. BERT Mutation outperforms all other mutation operators with a clear improvement in solution quality and convergence speed. This trend persists across other problem instances.

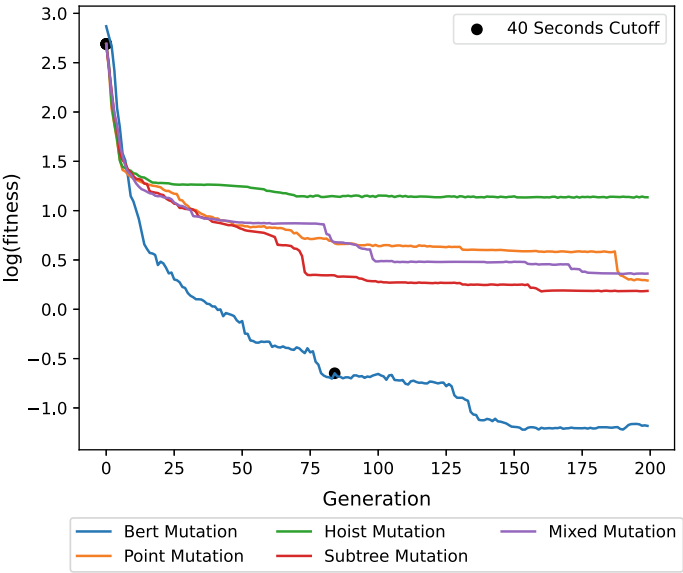


Fig. 3.7 Fitness value of best individual versus generation, of each mutation operator for the `non_analytic` dataset. The fitness value is averaged over 10 runs

As with DNC, the figure also shows the trade-off between the runtime and the achieved solution quality. The cutoff times are denoted in the graphs in Figure 3.7. After 40s only, our BERT mutation is far better than the baselines.

In Table 3.2, we compare the time per generation for each operator. While our operator is slightly slower than the baseline operators, the trade-off between runtime and the quality of solutions is clear. Also, note that we are dealing with a roughly 0.2 s increase per generation, not weeks and months; thus, the added cost is probably worth it, given that a much better solution has been obtained.

The work on the BERT operator is still ongoing. We are currently evaluating it in non-regression domains. Moreover, the operator is designed to use the individual as a string, whether the representation is a GP tree or a GA vector. Therefore, we anticipate

Table 3.2 Generation times (in seconds) of each operator

Operator	Mean	Std	Max
BERT mutation	0.328	0.145	0.672
Hoist mutation	0.145	0.061	0.307
Mixed mutation	0.146	0.062	0.325
Point mutation	0.145	0.062	0.310
Subtree mutation	0.139	0.066	0.329

that it will outperform baseline GA operators as well, although this evaluation is still in progress.

3.4 Discussion

During the mask replacement process, we chose to fill in the missing masked nodes using a Depth-First Search (DFS) approach. This assumption may not be optimal and could potentially be improved upon. This is a crucial step since the replacement of a single node affects all subsequent replacements, making the order of replacements extremely significant. Other approaches to traverse the masks include an unrolled traversal order, which could simplify weight assignment to important operators.

We also contemplated replacing online learning with a pre-training approach, where we would sample in smart ways from the population. However, this approach presents significant challenges. The operators need to be versatile enough to support any direction the population decides to explore in the search space. Consequently, the sampling would need to be very extensive—and thus expensive. We determined that it is more effective to implement online learning, which adapts naturally to the dynamic changes in the population.

It is important to note a crucial difference between the operators. The DNC operator does not maintain states between generations, meaning that at any given application, it chooses only between the current two parents and cannot introduce new variability into the population. On the other hand, the BERT mutation operator does maintain states between generations and can reintroduce genes to the population that worked well in earlier generations but were lost through evolution.

3.5 Concluding Remarks

We presented two methods employing the power of deep learning to improve an evolutionary algorithm’s genetic operators. Both the deep learning-based crossover and the deep learning-based mutation significantly outperformed their competition.

Note that the BERT mutation is perhaps unique in its being representation independent.

Our future work comprises two main avenues, notably, testing and extending deep neural crossover to other domains, and validating our preliminary results on the BERT mutation through extensive evaluation.

Acknowledgements This research was partially supported by the Israeli Council for Higher Education (CHE) via the Data Science Research Center, Ben-Gurion University of the Negev, Israel. Eliad Shem-Tov was supported by the Negev fellowship.

References

1. Ahmed, Z.H.: Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. *Int. J. Biomet. Bioinform. (IJBB)* **3**(6), 96 (2010)
2. Akter, S., Nahar, N., ShahadatHossain, M., Andersson, K.: A new crossover technique to improve genetic algorithm and its application to TSP. In: 2019 International Conference on Electrical, Computer and Communication Engineering (ECCE), pp. 1–6. IEEE (2019). <https://doi.org/10.1109/ECACE.2019.8679367>
3. 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. (1998)
4. Bello, I., Pham, H., Le, Q.V., Norouzi, M., Bengio, S.: Neural combinatorial optimization with reinforcement learning (2016). arXiv preprint [arXiv:1611.09940](https://arxiv.org/abs/1611.09940)
5. Blanchard, A.E., Shekar, M.C., Gao, S., Gounley, J., Lyngaas, I., Glaser, J., Bhowmik, D.: Automating genetic algorithm mutations for molecules using a masked language model. *IEEE Trans. Evol. Comput.* **26**(4), 793–799 (2022)
6. Brooks, T., Pope, D., Marcolini, M.: Airfoil self-noise. UCI Machine Learning Repository (2014). <https://doi.org/10.24432/C5VW2C>, <https://doi.org/10.24432/C5VW2C>
7. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: pre-training of deep bidirectional transformers for language understanding (2018). arXiv preprint [arXiv:1810.04805](https://arxiv.org/abs/1810.04805)
8. Eiben, A.E., Raue, P.E., Ruttkay, Z.: Genetic algorithms with multi-parent recombination. In: *International Conference on Parallel Problem Solving from Nature*, pp. 78–87. Springer (1994)
9. Eiben, A.E., Smith, J.E.: *Introduction to Evolutionary Computing*. Springer (2015)
10. Friedman, J.H.: Multivariate adaptive regression splines. *Ann. Stat.* **19**(1), 1–67 (1991)
11. gplearn github. <https://github.com/trevorstephens/gplearn>
12. Johnson, D.S., Trick, M.A.: Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, October 11–13, 1993, vol. 26. American Mathematical Society (1996)
13. Joshi, M., Chen, D., Liu, Y., Weld, D.S., Zettlemoyer, L., Levy, O.: Spanbert: improving pre-training by representing and predicting spans. *Trans. Assoc. Comput. Linguist.* **8**, 64–77 (2020)
14. Kiraz, B., Bidgoli, A.A., Ebrahimpour-Komleh, H., Rahnamayan, S.: A novel collective crossover operator for genetic algorithms. In: 2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 4204–4209. IEEE (2020)
15. Lehman, J., Gordon, J., Jain, S., Ndousse, K., Yeh, C., Stanley, K.O.: Evolution through large models. In: *Handbook of Evolutionary Machine Learning*, pp. 331–366. Springer (2023)
16. Liu, H., Zong, Z., Li, Y., Jin, D.: Neurocrossover: an intelligent genetic locus selection scheme for genetic algorithm using reinforcement learning. *Appl. Soft Comput.* **146**, 110,680 (2023)
17. Livne, A., Tov, E.S., Solomon, A., Elyasaf, A., Shapira, B., Rokach, L.: Evolving context-aware recommender systems with users in mind. *Expert Syst. Appl.* **189**, 116,042 (2022)
18. Magalhaes-Mendes, J.: A comparative study of crossover operators for genetic algorithms to solve the job shop scheduling problem. *WSEAS Trans. Comput.* **12**(4), 164–173 (2013)
19. McKay, B., Willis, M.J., Barton, G.W.: Using a tree structured genetic algorithm to perform symbolic regression. In: *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, pp. 487–492. IET (1995)
20. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in Pytorch (2017)
21. Poli, R., Langdon, W.B.: Schema Theory for Genetic Programming with One-Point Crossover and Point Mutation. *Evol. Comput.* **6**(3), 231–252 (1998). <https://doi.org/10.1162/evco.1998.6.3.231>
22. Schoenfeld, J.E.: Fast, exact solution of open bin packing problems without linear programming. Draft, US Army Space and Missile Defense Command, Huntsville, Alabama, USA (2002)

23. Semenkin, E., Semenkina, M.: Self-configuring genetic algorithm with modified uniform crossover operator. In: *Advances in Swarm Intelligence: Third International Conference, ICSI 2012, Shenzhen, China, June 17–20, 2012 Proceedings, Part I 3*, pp. 414–421. Springer (2012)
24. Shem-Tov, E., Elyasaf, A.: Deep neural crossover. In: *Proceedings of the Genetic and Evolutionary Computation Conference (to be published), GECCO'24*. Association for Computing Machinery (2024). <https://arxiv.org/abs/2403.11159>
25. Uy, N.Q., Hoai, N.X., O'Neill, M.: Semantics based mutation in genetic programming: the case for real-valued symbolic regression. In: *15th International Conference on Soft Computing, Mendel*, vol. 9, pp. 73–91 (2009)
26. Van Belle, T., Ackley, D.H.: Uniform subtree mutation. In: *European Conference on Genetic Programming*, pp. 152–161. Springer (2002)
27. Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. *Adv. Neural Inform. Process. Syst.* **28** (2015). https://proceedings.neurips.cc/paper_files/paper/2015/file/29921001f2f04bd3baee84a12e98098f-Paper.pdf
28. Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.* **8**, 229–256 (1992)
29. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al.: Huggingface's transformers: state-of-the-art natural language processing (2019). arXiv preprint [arXiv:1910.03771](https://arxiv.org/abs/1910.03771)
30. Yeh, I.C.: Concrete compressive strength. UCI Machine Learning Repository (2007). <https://doi.org/10.24432/C5PK67>

Chapter 4

Survey of Genetic Programming and Large Language Models



Erik Hemberg, Steven Jorgensen, and Una-May O'Reilly

Abstract There now exist algorithms that use both Genetic Programming (GP) and Large Language Models (LLMs) to evolve solutions. This is a composition of two different methods that show some potential. We survey GP and LLM algorithms and their applications from a GP centric perspective. We provide a taxonomy, a breakdown of application domains, and a categorization of the implementation details. Finally, we discuss existing research gaps and possible directions for future work.

Keywords Large language models · Genetic programming

4.1 Introduction

Large language models (LLMs) are Artificial Intelligence systems. An LLM works in the input-output space of natural language, i.e. unstructured text. It processes natural language prompts and responds in natural language. Technically speaking, a Language Model is often a pre-trained transformer model [39] which has distilled patterns of statistical associations within a training corpus. Pre-training back-propagates errors from predictions that complete text sequences from the corpus. Often the LLM is further fine-tuned on selected data. It can then perform approximate retrieval of these patterns to respond to input sequences. There are various fine-tuning methods, and one important application is general instruction-following. E.g. the prompt-response is improved with Reinforcement Learning with Human Feedback [9].

In comparison, Genetic Programming (GP), inspired by Neo-Darwinian evolution, operates on a population of candidate solutions of variable size [23]. Generally,

E. Hemberg (✉) · S. Jorgensen · U.-M. O'Reilly
ALFA, MIT CSAIL, Cambridge, MA, USA
e-mail: hembergerik@csail.mit.edu

S. Jorgensen
e-mail: st28850@mit.edu

U.-M. O'Reilly
e-mail: unamay@csail.mit.edu

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2025
S. M. Winkler et al. (eds.), *Genetic Programming Theory and Practice XXI*, Genetic and Evolutionary Computation, https://doi.org/10.1007/978-981-96-0077-9_4

a GP system uses operators to optimize these solutions. It has one operator that initializes a population of candidate solutions, two that iteratively test each solution and calculate its fitness, one that selects parents by referencing solution fitness, multiple operators that randomly vary parental copies, and one to compose a new population that replaces the old. A basic GP is set up with its operators. Before a run, it is provided with a solution representation and a fitness function.

LLMs can generate code to solve many tasks. GP has been applied to, e.g. software engineering tasks and even program synthesis [20]. Both GP and LLMs *can evolve code*, i.e. perform program synthesis with an evolutionary-inspired method. For example, an LLM performs the function of different operators in the OpenELM Library [4], an LLM does code-level neural architecture search [7], and, without a typical GP algorithm but by using *similar* evolutionary mechanisms, some GP benchmarks for program synthesis have been solved with an LLM-supported procedure [20]. Figure 4.1 shows an overview of GP with LLM, i.e. how the general algorithmic structure of GP and evolutionary operators can use an LLM to evolve solutions.

A position paper about leveraging Foundational Models for Black-Box Optimization highlights that GP is very flexible since it can have both discrete and continuous parameters, multi-step constraints on the parameters, and both bounded and unbounded size [35]. There is one existing survey about evolutionary computation and large language models [41]. In this contribution, we extend this survey by providing a more GP focused taxonomy. In addition, we provide further attribution of properties to an updated collection of citations, as well as a discussion. Furthermore, our survey also complements the more practical investigation and discussion regarding evolving code with a large language model [12].

The goal of this contribution is to:
provide a taxonomy of GP and LLM research,
survey existing GP and LLM investigations, and
discuss a possible roadmap and future work

LLMs are parametric models that use data to set its parameters. GP is not a parametric model, instead data is often used to evaluate the quality of generated solutions. Figure 4.2 shows an overview of observed uses of GP and LLM in this paper. We make two distinctions. GP or LLM is used *as part of* the other during the

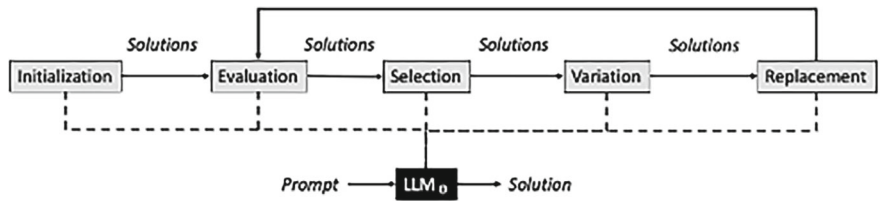


Fig. 4.1 Overview of a process with LLM as a part of GP in the form of operators. The population is called solutions. Prompts are different for each LLM operator

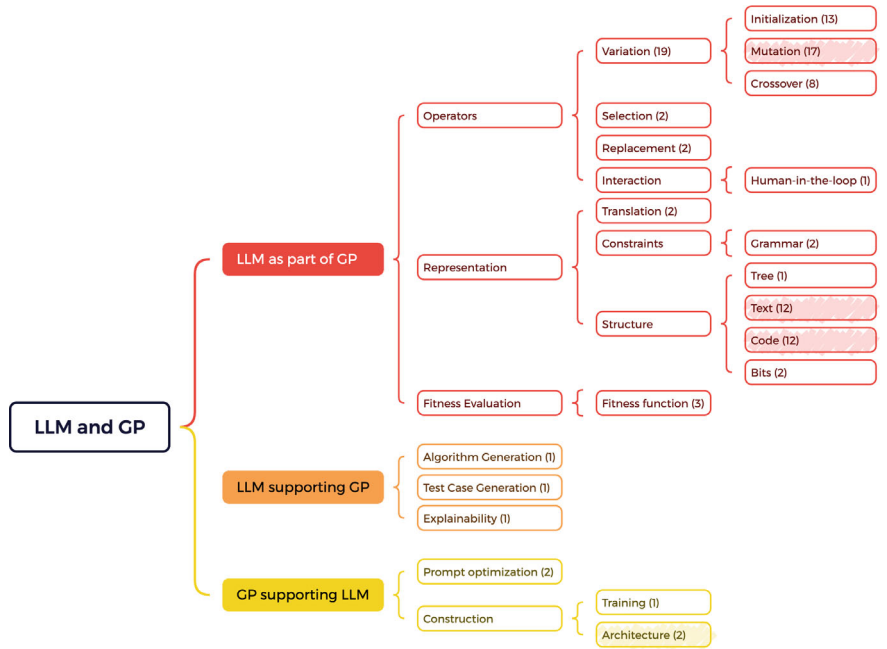


Fig. 4.2 Overview of GP and LLM uses. Numbers indicate papers in each category. Highlights indicate high number of papers

iterative process of GP evolution. GP or LLM is used *supporting* the other before or after the iterative process of GP evolution or LLM training. A brief list of examples and categories is as follows (in Sect. 4.3 we elaborate further):

- E.g. a GP system that uses an LLM as the mutation operator [24]. Different sub-categories are operators, representation, and fitness evaluation.
- E.g. GP for prompt optimization [8]. Different sub-categories are prompt optimization and artificial neural network construction.
- E.g. LLM for test case generation [13]. Different sub-categories are algorithm creation, test case generation, and explainability.

We start, in Sect. 4.2, with background on LLMs. In Sect. 4.3, we describe and contrast GP and LLM work. We present a discussion and roadmap in Sect. 4.4. Finally, in Sect. 4.5, we conclude with a summary of the paper’s contributions.

4.2 Background

For a parametric model like an LLM the training setup, architecture, domain, and the input formulation via a prompt are important properties. GP is a non-parametric model, so important attributes are operators, representation, and the domain where the solution quality is evaluated. Figure 4.3 shows some GP and LLM attributes we consider.

In Section 4.2.1, we briefly present Large Language Models. In Sect. 4.2.2, we present GP with LLM operators.

4.2.1 Large Language Models

Architecture

Language models (LM) generatively model the statistical likelihood of a corpus of text [19], implying they can generate text completions using what can be described as approximate matching between prompt text and text observed during training. This capability makes them very useful for natural language tasks such as translation, summarization, or text classification. The capabilities of language models abruptly accelerated with the adoption of transformer architectures [39]. Transformer architectures avoid the constraints of prior language models and now allow vast amounts of computation to be marshaled for training with parallelization. They also lever-

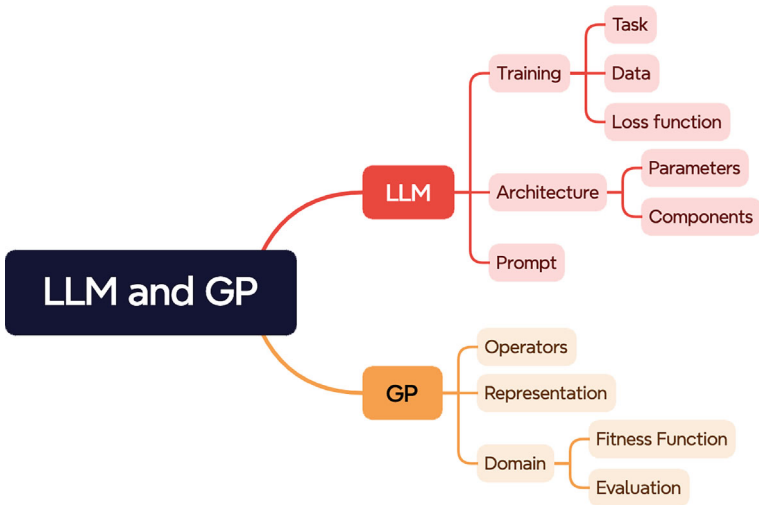


Fig. 4.3 Overview of some GP and LLM attributes

age training on the task of token unmasking or sequence completion which allows training on massive unlabeled text corpuses such as the Internet.

The “Large” in “Large Language Models” typically refers to language models with at least 1M parameters. LLMs catapulted to global attention and wide-spread adoption in 2022 with the introduction of OpenAI’s “GPT” series (Generative Pre-trained Transformer, [5, 26, 29]) which ranged from 120M to 175B parameters and used training sets that ranged in size from 1B [29] to 300B [5].

LLM behavior can be guided in multiple ways:

LLMs have a temperature parameter. A low temperature makes prompt completion more deterministic. Raising the temperature leads to more variability in prompt completion. Some approaches raise the LLM temperature to obtain more diverse solutions [7].

The LLM parameters are updated based on domain-specific data and a training procedure, resulting in Θ' where $\Theta' = g(X, Y, \Theta)$, Θ are the model parameters, $X \in \mathcal{T}$ is the input, $Y \in \mathcal{T}$ is the output. Note that when training data is generated by the model and the updated and fine-tuned model is used, the algorithm is self-evolving [38].

Prompt

ChatGPT made big impact due to an easy-to-use question-answer interface. This interface (and API) accepts from some actor, a natural language text sequence, which functions to the actor as a query, question, or “*prompt*”. The actor assigns knowledge, meaning, and intention to the text in the prompt. The actor can provide problem context, problem solution examples, chain-of-thought reasoning, or other information, to aid the LLM’s generative pattern matching and completion.

Prompt engineering is a term used to describe the manipulation of input text to achieve desired outputs. Popular prompt engineering techniques include the following:

A simple predefined prompt $\rho = T$, $T \in \mathcal{T}$.

A human interacts with the LLM to manipulate the prompts and responses, $\rho', y' = H(\rho, f(\rho|\Theta))$.

The prompt is a template that is expanded at run-time using run-time information, $\rho = \rho(\mathbf{x})$.

An extension of Template providing examples of correct responses, $\rho = \rho(\mathbf{x}, X, Y)$, $|X| = |Y|$, $X, Y \in \mathcal{T}$. Note, this is sometimes also called in-context learning.

A sequence of LLM calls, $y_i = f(\rho_{i-1}(y_{i-1}))$, $y_0 = f(\rho_0())$, $i \in \mathbb{Z}^+$. For example, chain-of-thought is a type of chaining that does not presume any external environment changes [44].

A combination of Template and Chaining ($i = 2$) where an article is first summarized and then provided as input to a template.

A prompt’s content is (externally) optimized according to some function u , $\rho' = g(\rho, u(f(\rho|\Theta)))$, $u : \mathcal{T} \rightarrow \mathbb{R}$, $v = u(\rho)$.

Task Completion

Of interest to genetic programming is how LLMs can perform several types of coding tasks, e.g. automatic bug fixing per [33], although they do not work in an evolution-inspired manner. Language models that perform program synthesis and transformer models like Github Copilot specifically trained for code tasks have been released. For example, Meta released the Code Llama family of models, with LLMs specifically pre-trained and tuned for coding in August 2023 by Meta [32]. In benchmark testing with Code Llama outperformed state-of-the-art publicly available LLMs on code tasks [32]. In 2022, [34] evaluated [GitHub Copilot](#) on standard GP-community program synthesis benchmark problems. They compared the Copilot’s results to reported genetic programming state of the art and observed that the performance of the two approaches was quite similar.

4.2.2 A LLM as Part of GP Algorithm

Algorithm 4.1 describes a possible LLM as part of GP [12] implementation. It takes a general approach to using an LLM—every one of its operators uses an LLM; many variants of LLM as part of GP use fewer. Algorithm 4.1 exhibits several features that make it quite dissimilar to GP. The first difference is that the unit of evolution, code, is represented as a variable-length sequence of text. This sequence is *executable but it is not a parse-tree*. The evolutionary operators in Algorithm 4.1 utilize a LLM to: initialize a population of candidate genotypes (code, line 1), execute and evaluate them (lines 3 and 4), select parents (line 7), create children with variation (lines 8 and 10), and replace the old population with the old one (line 12). Finally, Algorithm 4.1 includes an LLM-based operator for designating the run’s solution (line 13).

An LLM as part of GP operator requires three steps:

1. Formulate: Compose the prompt via calling the operator’s prompt-function.
2. Interface: Send the prompt to the LLM and collect the LLM’s response.
3. Check: Ensure response is well formed.

A significant parameter of each operator is its prompt, ρ . A *prompt variable or object* is a sequence of text that can be decomposed into ordered elements, e.g.:

$\langle \rho \rangle ::= \langle \text{EXAMPLES} \rangle \langle \text{QUERY} \rangle \langle \text{PRIMITIVES} \rangle \langle \text{RESPONSE_FORMAT} \rangle$,
where ρ is a sequence of text $\rho \in T^m$.

E.g. Generate a mathematical expression. Use the listed symbols
[+, -, *]. Provide no additional text in response. Format
output in JSON as {"expression": "<expression>"}

Some prompts contain problem-dependent information. For example, the prompts for initialization and mutation can include the problem primitives. Some prompts are templates with elements that need to be inserted at run-time. The three major preparatory steps ahead of an LLM as part of GP run are as follows:

Algorithm 4.1: LLM as part of GP Operators: i_{LLM} (Initialization), e_{LLM} (Evaluation), ϕ_{LLM} (Fitness measurement), s_{LLM} (Selection), v_{xoLLM} (Crossover), v_{muLLM} (Mutation), r_{LLM} (Replacement), b_{LLM} (Designate best solution)

Input : RUN HYPER-PARAMETERS

g : Generations, n : Population size,
LLM : LLM,

PROMPT-FUNCTIONS (HYPER-PARAMETERS)

$\rho_i(F \cup T)$: Prompt for initialization,

$\rho_e(\mathbf{a})$: Prompt for evaluation,

$\rho_\phi(\mathbf{a})$: Prompt for fitness measure,

$\rho_s(\mathbf{a},)$: Prompt for selection,

$\rho_{vxo}(\mathbf{a})$: Prompt for crossover,

$\rho_{vmu}(\mathbf{a})$: Prompt for mutation,

$\rho_r(\mathbf{a})$: Prompt for replacement,

$\rho_b(\mathbf{a})$: Prompt for picking best solution,

Return: P^* : best solution

```

1  $P \leftarrow i_{LLM}(n, \rho_i \leftarrow \rho_i(F \cup T))$  //Random initialization of population
  //Iterate over generations
2 for  $i \in [1, \dots, g]$  do
3    $\mathbf{y} \leftarrow e_{LLM}(\rho_e \leftarrow \rho_e(P, D, E))$  //Evaluate solution
4    $\mathbf{f} \leftarrow \phi_{LLM}(\rho_\phi \leftarrow \rho_\phi(\mathbf{y}, D))$  //Measure the fitness of the solution
5    $P' \leftarrow \emptyset$  //New population
  //Iterate over population
6   while  $|P'| \leq n$  do
7      $p_k, p_l \leftarrow s_{LLM}(P, \rho_s \leftarrow \rho_s(P))$  //Select two parents
8      $p'_k, p'_l \leftarrow v_{xoLLM}(\rho_{vxo} \leftarrow \rho_{vxo}(p_k, p_l))$  //Variation with crossover
    //Iterate over children
9     for  $j \in [1, \dots, n]$  do
10       $p' \leftarrow v_{muLLM}(\rho_{vmu} \leftarrow \rho_{vmu}(F \cup T, p'))$  //Variation with mutation
11       $P' \leftarrow P' \cup \{p'_k, p'_l\}$  //Add to new population
12   $P \leftarrow r_{LLM}(\rho_r \leftarrow \rho_r(P'))$  //Replace the population
13 return  $P^* \leftarrow b_{LLM}(P, \rho_b())$  //Best solution

```

1. Deciding the language that will express the candidate solutions plus problem-dependent primitives and any primitives built-in to the language to be used.
2. Selecting the prompt-functions of all operators implemented using an LLM.
3. Choosing the hyper-parameters for controlling the run, including the termination criterion.

LLM as part of GP versus GP

We summarize GP and LLM as part of GP differences in Table 4.1 from [12]. Both GP and LLM as part of GP are EAs where the evolving unit is code. Both execute with the same procedural logic and operators of an EA. Both represent code with symbols naming problem primitives. However, GP relies upon the code being represented by a structure. This allows both code execution and explicit manipulation, through variation operators, of blind genotype variation. For example, Koza-style GP works with code in a parse-tree structure.

Table 4.1 Comparison of GP and LLM as part of GP on different aspects of GP search

Basis of comparison	GP	LLM as part of GP
Environment	Program execution environment	Program execution environment and LLM which is a generative pattern completion system using token sequence pattern-matching
Run of a system	A GP run executes procedural software where the code is data, the operators work on code structure, and the code is bespoke evaluated and assigned numerical fitness	A LLM as part of GP run executes procedural software that composes text-based NL prompts, interacts with an LLM, and collects responses
Representation	Genotype/phenotype is a data structure with structural properties and executability	Code is token sequence with implicit structural properties, pattern-matching, and bias within the LLM
Variation	Structural, blind to meaning	Not structural, blind to user beyond prompt content
Selection/replacement	Comparative, based on numeric ranking	Comparative, prompt can task the opaque LLM to rank
Evaluation	Uses bespoke execution environment in a general-purpose program execution environment	Uses a general-purpose program execution environment
Fitness	Numeric-based	Numeric or expressed with natural language

In contrast LLM as part of GP works with a code snippet that is tokenized for input to a LLM. Unlike GP, LLM as part of GP forgoes explicit structural manipulation of the code during variation. Instead, it tasks the LLM to recombine, or mutate code by using a prompt, and the only control it exercises over how the operation is accomplished is through the prompt formulation. The LLM is a complex, pre-trained, approximate pattern-based sequence completion system; effectively a black box [17, 40].

4.3 GP and LLM Work

The topic of GP intersecting with LLMs relatively new so we present a survey that includes both peer-reviewed papers and non-peer-reviewed papers found on <https://arxiv.org> at the time of writing. Table 4.2 identifies contributions by its citation in this article, title, and problem domain. Figure 4.4 shows an overview of identified categories of uses for GP and LLM.



Fig. 4.4 Overview of GP and LLM domains and sub-domains

In Sect. 4.3.1, we survey GP and LLM work based on problem domain and use. In Sect. 4.3.2, we survey the GP and LLM work based on some implementation properties.

4.3.1 By Problem Domain and Use of GP and LLM

Figure 4.4 gives an overview of the GP and LLM domains.

We first group related papers by problem domain. We observe an array of domains that we group as

Program Synthesis, e.g. code generation [20] and symbolic regression [24].

Neural Networks, e.g. neural architecture search (NAS) [7].

Design, e.g. game design [14].

Agent Controllers, e.g. agent-based controllers [15].

Text generation, e.g. explanation of GP trees [22].

Boolean problems, e.g. one-max [4].

Table 4.2's third column identifies the surveyed paper's problem domains. Columns four to six in Table 4.2 categorize how GP and LLMs are used. We now describe the works based on these categories.

LLM as part of GP Within the LLM as part of GP group there are papers that evolve agent controllers or their reward functions [15, 21], solve program synthesis, symbolic regression, and Boolean parity problems [4, 20, 24].

To fine-tune a LM for a particular context: game terrain, [15] introduce an approach called Evolution through Large Models (ELM). Also, the earliest paper with an evolutionary operator using an LLM, this work uses an LLM-based mutation operator and a Quality-Diversity to generate training examples for the fine-tuning. To support agent-based Reinforcement Learning, EUREKA [21] performs LLM-based variation to evolve the agent's reward function. The approach uses LLM-created reward function variations that are based on reflective summarizations of the agent's performance. It also allows human interaction and feedback on the reward function.

Table 4.2 Overview of work intersecting LLM and GP. The work is categorized as LLM as part of GP, GP supporting LLM, and LLM supporting GP

Author	Title	Problem	LLM as part of GP	GP supporting LLM	LLM supporting GP
[31]	Mathematical discoveries from program search with large language models	Program synthesis	✓		
[20]	Fully autonomous programming with large language models.	Program synthesis	✓		
[4]	The openelm library: Leveraging progress in language models for novel evolutionary algorithms.	Code for agent controller, boolean parity, program synthesis, Text	✓		
[24]	Language model crossover: variation through few-shot prompting	Code for agent controller, symbolic regression, boolean parity, text	✓		
[28]	Enhancing large language models-based code generation by leveraging genetic improvement	Program synthesis	✓		
[21]	Eureka: Human-level reward design via coding large language models	Code for reward function	✓		
[7]	Evoprompting: language models for code-level neural architecture search	Code for neural architecture search	✓		
[25]	Neural architecture search via large language models and quality-diversity optimization.	Code for neural architecture search	✓		
[14]	Chatgpt and other large language models as evolutionary engines for online interactive collaborative game design	Text for game design	✓		
[16]	SCAPE: Searching Conceptual Architecture Prompts using Evolution	Text for architecture images	✓		
[37]	Program synthesis with generative pre-trained transformers and grammar-guided genetic programming grammar	Text for code	✓		
[6]	Enhancing genetic improvement mutations using large language models	Program improvement	✓		
[36]	MarioGPT: Open-ended Text2Level generation through large language models	Text for game design	✓		

(continued)

Table 4.2 (continued)

Author	Title	Problem	LLM as part of GP	GP supporting LLM	LLM supporting GP
[15]	Evolution through large models.	Code for agent controller	✓	✓	
[8]	Promptbreeder: self-referential self-improvement via prompt evolution	Prompt Search	✓	✓	
[10]	Connecting large language models with evolutionary algorithms yields powerful prompt optimizers	Prompt search		✓	
[42]	Wizardlm: empowering large language models to follow complex instructions	Data for LLM tuning		✓	
[18]	Algorithm evolution using large language model	Code for optimization algorithm	✓		✓
[45]	Self-taught Optimizer (STOP): Recursively self-improving code generation	Code for optimization			✓
[13]	Large language model-based test case generation for GP agents	Test cases for GP			✓
[22]	Explaining genetic programming trees using large language models	Explaining GP trees			✓

For program synthesis, Romera-Paredes et al. [31] introduce a distributed (island) evolutionary method using an LLM for searching the function space of programs they call FunSearch. A prompt that generates a new program is constructed by combined multiple programs ordered by fitness. Their FunSearch method searches for programs that describe how to solve a problem, instead of a direct solution. On an algorithmic problem, online bin packing they find new program heuristics that improve on baselines. In addition, there is work on PSB2 benchmark [11]. Liventsev et al. [20] introduce an approach to program synthesis called SEIDR: Synthesize, Execute, Instruct, Debug, and Rank. First, program snippet solutions are LLM-generated (Synthesize). Second, with extra program context, every snippet is externally executed and assigned a fitness (Execution). Next, some solutions are selected based on a top- k ranking (Rank). A repair phase (Instruct) then tries to analyze these failed solutions by considering their performance (Debug). To balance exploitation (repairing current solutions) and exploration (replacing the current solution), a beam-search algorithm is used. On the PSB2 SEIDR achieves the state-of-the-art result with 19 solved problems out of 25 with under 1000 program executions. Another example is [28] who improve the code generated by an LLM. They use Grammatical Evolution and automatically specialize a grammar based on the output of the LLM. On 25 PSB2

problems and with 5 different LLMs they can improve the code generated by LLMs. References [37] has a similar approach also on PSB2.

A variety of typical simple GP problems (e.g. agent controllers, Symbolic Regression, OneMax) are investigated by [4]. They introduce OpenELM, an open-source Python library for designing evolutionary algorithms that leverage LLMs. Many of the same problems are investigated by [24]. They input text-based genotypes that are either code, plain-text sentences, or equations to the LLM. They then interpret and use the corresponding LLM responses as those genotypes' offspring. Their experiments highlight the versatility of LLM-based crossover and the success of the LLM-based approach.

Design is another problem domain where typically the genotype representation is text. Lanzi et al. [14] present a collaborative game design framework that combines interactive evolution and LLMs to simulate the typical human design process. They use interactive evolution for selection and LLMs for a recombination and variation of ideas. Another game design example is from [36] in which they use a fine-tuned GPT2 model to generate tile-based game levels. It is used for LLM-based initialization and variation of prompt text that describes a game level. For architectural design [16] proposed SCAPE a tool that combines evolutionary search with generative AI which enables exploration of creative and quality designs based on initial input in a simple GUI. SCAPE enables memory and vary prompts via text-based mutation and crossover. Based on independent architect assessment SCAPE improves image novelty and quality in a few iterations compared to DALLE-3.

NAS using a Python code representation is investigated by [7]. They use LLM-based mutation and crossover operators. For prompt engineering they use a few-shot learning technique, and they fine-tune with the sub-population that was not selected. In other NAS work [25] use an LLM and a Quality-Diversity algorithm to obtain variations of Python code defining a neural architecture. Their system LLMatic creates diverse networks that are high-performing.

GP supporting LLM We classify work that use GP to assist LLMs in the GP for LLM group. One example is prompt engineering, since hand-crafted prompts are often sub-optimal. Promptbreeder [8] mutates a population of text-based ask-prompts, evaluates them for fitness on a training set, and repeats this process over multiple generations to evolve task-prompts. In a self-adaptive way, Promptbreeder uses its mutation-prompts to improve the task-prompts. Another work on discrete prompt optimization uses LLM-based evolutionary operators to improve a population based on the development set [10]. Furthermore, Xu et al. [42] start with an initial set of instructions that they evolve and instruct the LLM to rewrite step by step into more complex instructions. The frequency of papers evolving prompts is likely due to the non-trivial nature of efficient prompt construction and the easy means with which an LLM handles text.

Finally, a meta-heuristic approach that is not explicitly described as GP, but that is similar in spirit, and which uses a LLM is introduced by [45]. An LLM is used to generate a scaffolding program that is iteratively improved.

LLM supporting GP. In the LLM for GP group we put research that use LLMs for aiding the use of GP. One example is [13] that use the LLM for improving GP with curriculum learning. They generate test cases of varying difficulty with the LLM, in the form of game levels and opponent strategies. These are then used to train a LGP system. Another example is [22] who use the LLM to improve the readability of GP solutions by explaining GP trees. Finally, Liu et al. [18] investigate the use of LLMs to evolve code for optimization algorithm. They also use LLM variation operators.

4.3.2 By Implementation

Next, across the surveyed papers, we focus on the implementation. We consider LLM-specific design decisions and specifications, which LLM-based evolutionary operators are Used, and what genotype representations are evolved.

Table 4.3 summarizes the surveyed papers based on their design choices, operators that use an LLM, and genotypes. **Operators Using an LLM** is second column of Table 4.3 which describes the surveyed papers' LLM operators. All papers use the LLM generative capabilities for variation, mutation, and crossover. A majority also use it for solution initialization, and a minority use it to measure fitness. **Genotypes** is the third column of Table 4.3 and it describes what kinds of genotype a paper evolves. Furthermore, we observe that a wide variety of prompt engineering techniques are used in the **LLM Design** column. The most popular choice is to use a Template. This is an obvious approach to integrating run-time information such as genotypes and fitness scores. We also include the LLM(s) used in the paper.

Finally, the last column **LLM** shows that the most frequently (9) used LLM is version of GPT-3.5. The second most popular LLM is GPT-4 (5). Both these models are proprietary and at the forefront of LLMs when they were released. There are also a wide array of open-source models (9) of varying sizes that are used. Some papers use fine-tuned models, but the majority do not do any fine-tuning.

4.4 Discussion and Future Work

In Sect. 4.4.1, we discuss risks and limits of LLM and GP. In Sect. 4.4.2, we present possible future work and opportunities.

4.4.1 Discussion

First we focus the discussion on risks and limitations [12]. Current LLMs have an impressive natural language interface. However, LLMs are *not capable of general human intelligence* or even language understanding. They are token-driven,

Table 4.3 Details of LLM operators, genotype, and prompt engineering. Design refers to LLM manipulation and prompt engineering. LLM Op. is the LLM operators

Cite	LLM Op.	Genotype	LLM design	LLM
<i>GP with LLM</i>				
[31]	Initialization, mutation	Code	Template,	Codey
[20]	Mutation, initialization	Code	Template, changing temperature, chaining, summarization	Code-davinci-edit-001, text-davinci-003
[15]	Mutation	Code	Template, fine-tuning	300M Diff model finetuned
[4]	Mutation, initialization, crossover, fitness measure (QD)	Code, text, bits	Template	CodeGen 350M, 2B, 6B Diff model finetuned, Pythia 12B, LLaMA 7B
[24]	Mutation, initialization, crossover, fitness measure (QD)	Code, text, bits	Few-shot	Pythia-70M, 160M, 410M, 1B, 1.4B, 2.8B, 6.9B, Pythia-1.4B, Pythia-2.8B, GALACTICA-1.3B, CodeGen-350M, 2B, 6B
[28]	Initialization	Code	Template	Alpaca 7B, Alpaca 13B, LLaMA 2 13B, ChatGPT, GPT4
[21]	Mutation	Code	Template, summarization, Human interaction	GPT-4
[7]	Mutation, initialization, crossover	Code text	Fine-tuning, few-shot, changing temperature	62B PALM
[25]	Mutation, initialization	Code	Zero shot, changing temperature	CodeGen-Mono 6B
[14]	Mutation, initialization, crossover	Text	Template, human interaction	GPT-3.5
[16]	Initialization, crossover, mutation	Text	Template	DALLE-4, GPT-4
[37]	Initialization	Text	Template	ChatGPT
[6]	Mutation	Code	Template	gpt-3.5-turbo
[36]	Mutation, initialization, crossover	Text	Zero Shot	MarioGPT (GPT-2, Finetuned)
<i>GP for LLM</i>				
[10]	Initialization, mutation, crossover	Text	Template	Alpaca-7B, text-davinci-003
[8]	Initialization, mutation	Text	Template, chaining	PaLM-2-L
[42]	Mutation	Text	Template	ChatGPT

(continued)

Table 4.3 (continued)

Cite	LLM Op.	Genotype	LLM design	LLM
LLM for GP				
[18]	Initialization, crossover, mutation	Code	Template	GPT-3.5, GPT-4
[45]	Mutation, fitness measure	Code	Optimization	gpt-4-0314, gpt-3.5-turbo-0613, Mixtral-8x7B-Instruct-v0.1
[13]	NA	Text	Zero Shot	GPT-3.5 MarioGPT
[22]	NA	GP Tree (Text)	RAG, template	gpt-3.5-turbo

token-outputting, generative pre-trained transformer models capable of approximate matching. Such a system cannot be trusted to consistently produce the right behavior, *in general* [1, 2]. Obviously, LLMs exhibit several unresolved issues, e.g. bias, accuracy, and resource requirements [3].

Risks of Using an LLM. The risk posed by LLM_GP’s operators is that the operator (and human designer) surrenders its control over the details of its behavior. LLM_GP also faces other risks and challenges:

A GP and LLM algorithm depends on the prompt composition. Further, LLMs lack many facets of general intelligence, while they appear to understand prompts. This leads to the risk of mistakenly assuming understanding when it does not exist. LLMs display biases based on their architecture, training dataset, and pre-training, e.g. token bias and positional bias. Additionally, an LLM is probabilistic and generative. In addition, performance is not consistent across LLMs. LLM pre-training is often not done by the LLM users and datasets for training and testing are frequently not well documented or shared. When an LLM is used via a model-provider’s API, experimental replicability may depend on whether a model release is preserved.

Collectively these risks directly and significantly impact the quality of the science that can be conducted with LLM and GP. In addition, working with LLMs is non-trivial and resource intensive.

Considerations for GP and LLM Investigations. GP publication standards and norms started from a primitive state and still evolve. Hemberg et al. [12] suggest some possible investigation standards.

Reporting

follow scientific standard for replicability;
report the exact model version along with its pre-training, its training data, and its fine-tuning.

Methods

*verify to what degree the model has been trained on the problem;
report computational effort for the entire method.*

Integrity

*be responsible with environmental cost;
use the LLM ethically and keep usage aligned with human values.*

4.4.2 Future Work and Opportunities

Figure 4.5 gives an overview of the GP and LLM discussion and future work. The main directions are as follows:

New problem domains to apply GP and LLM to, e.g. further exploration of open ended problems as proposed by [15]. In addition, domains in science and engineering could be further explored with GP and the ability to incorporate domain knowledge with LLMs.

There are numerous opportunities to incorporate LLM operators in GP. E.g. using insights from LLM interpretability [30] to guide LLM operators. The area of GP and coevolution [27] can use LLMs. Efficiency could be improved as demonstrated by [13]. Naturally, there is inspiration to draw from biology.

Both GP and LLMs are complex and analysis methods are needed, e.g. can explainability be improved for GP with LLM as in [22]. The safety of LLMs is a wide open area [3], which will also apply to GP and LLM systems. Another important area is security as shown by evading classifiers [43].

There are limited studies on using GP with LLM, e.g. for generating diverse responses.

4.5 Summary

There exist algorithms that use both Genetic Programming (GP) and Large Language Models (LLMs) to evolve solutions. This is a composition of two different methods, parametric and non-parametric, that show some potential. We surveyed GP and LLM algorithms and their applications from a GP centric perspective. We provide a taxonomy, based on LLM as part of GP, GP supporting LLM, and LLM supporting GP. We also showed a breakdown of application domains and a categorization of the implementation details.

Finally, we discussed risks, existing research gaps, and possible directions for future work. Using an LLM to implement evolutionary operators provides new capabilities, but also incurs new costs: in a run the time to interact with the LLM for each prompt and the cost of prompting can be significant. As well, a hidden, but significant cost is the pre-training of the LLM.

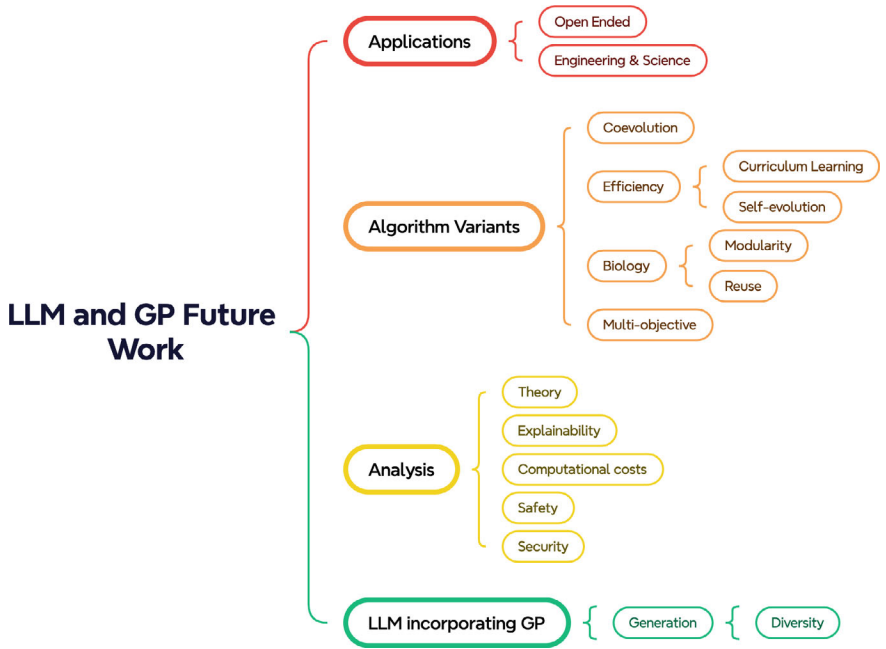


Fig. 4.5 Overview of some GP and LLM discussion, opportunities, and future work

Acknowledgements DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the U.S. Air Force. ©2024 Massachusetts Institute of Technology. Delivered to the U.S. Government with Unlimited Rights, as defined in DFARS Part 252.227-7013 or 7014 (Feb 2014). Notwithstanding any copyright notice, U.S. Government rights in this work are defined by DFARS 252.227-7013 or DFARS 252.227-7014 as detailed above. Use of this work other than as specifically authorized by the U.S. Government may violate any copyrights that exist in this work. Work is also supported by US Government Contract No. FA8075-18-D-0008.

References

1. Connectionists: Chomsky’s apple. <https://mailman.srv.cs.cmu.edu/pipermail/connectionists/2023-March/039546.html>. Accessed 27 Oct. 2023
2. On evaluating understanding and generalization in the arc domain. <https://aiguide.substack.com/p/on-evaluating-understanding-and-generalization>. Accessed 27 Oct. 2023
3. Bereska, L., Gavves, E.: Mechanistic interpretability for AI safety—a review (2024). <https://api.semanticscholar.org/CorpusID:269293418>
4. Bradley, H., Fan, H., Galanos, T., Zhou, R., Scott, D., Lehman, J.: The openelm library: leveraging progress in language models for novel evolutionary algorithms. In: Genetic Programming Theory and Practice XX. Springer (2024)

5. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language models are few-shot learners (2020)
6. Brownlee, A.E.I., Callan, J., Even-Mendoza, K., Geiger, A., Hanna, C., Petke, J., Sarro, F., Sobania, D.: Enhancing genetic improvement mutations using large language models. In: International Symposium on Search Based Software Engineering (2023). <https://api.semanticscholar.org/CorpusID:264803513>
7. Chen, A., Dohan, D., So, D.: Evoprompting: language models for code-level neural architecture search. *Adv. Neural Inform. Process. Syst.* **36** (2024)
8. Fernando, C., Banarse, D., Michalewski, H., Osindero, S., Rocktäschel, T.: Promptbreeder: self-referential self-improvement via prompt evolution (2023)
9. Griffith, S., Subramanian, K., Scholz, J., Isbell, C.L., Thomaz, A.L.: Policy shaping: integrating human feedback with reinforcement learning. *Adv. Neural Inform. Process. Syst.* **26** (2013)
10. Guo, Q., Wang, R., Guo, J., Li, B., Song, K., Tan, X., Liu, G., Bian, J., Yang, Y.: Connecting large language models with evolutionary algorithms yields powerful prompt optimizers (2023)
11. Helmuth, T., Kelly, P.: Applying genetic programming to psb2: the next generation program synthesis benchmark suite. *Genet. Program. Evol. Mach.* **23**(3), 375–404 (2022)
12. Hemberg, E., Moskal, S., O'Reilly, U.M.: Evolving code with a large language model (2024). *ArXiv* [arXiv:2401.07102](https://arxiv.org/abs/2401.07102), <https://api.semanticscholar.org/CorpusID:266999468>
13. Jorgensen, S., Nadizar, G., Pietropoli, G., Manzoni, L., Medvet, E., O'Reilly, U.M., Hemberg, E.: Large language model-based test case generation for GP agents. In: Proceedings of the Genetic and Evolutionary Computation Conference. Association for Computing Machinery, New York, NY, USA (2024)
14. Lanzi, P.L., Loiacono, D.: Chatgpt and other large language models as evolutionary engines for online interactive collaborative game design (2023). *ArXiv preprint* [arXiv:2303.02155](https://arxiv.org/abs/2303.02155)
15. Lehman, J., Gordon, J., Jain, S., Ndousse, K., Yeh, C., Stanley, K.O.: Evolution through large models. In: Handbook of Evolutionary Machine Learning, pp. 331–366. Springer (2023)
16. Lim, S.L., Bentley, P.J., Ishikawa, F.: Scape: Searching conceptual architecture prompts using evolution (2024). *ArXiv* [arXiv: 2402.00089](https://arxiv.org/abs/2402.00089), <https://api.semanticscholar.org/CorpusID:267365081>
17. Lipkin, B., Wong, L., Grand, G., Tenenbaum, J.B.: Evaluating statistical language models as pragmatic reasoners (2023)
18. Liu, F., Tong, X., Yuan, M., Zhang, Q.: Algorithm evolution using large language model (2023). *ArXiv* [arXiv: 2311.15249](https://arxiv.org/abs/2311.15249), <https://api.semanticscholar.org/CorpusID:265456254>
19. Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., Neubig, G.: Pre-train, prompt, and predict: a systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.* **55**(9), 1–35 (2023)
20. Liventsev, V., Grishina, A., Härmä, A., Moonen, L.: Fully autonomous programming with large language models (2023). *ArXiv preprint* [arXiv:2304.10423](https://arxiv.org/abs/2304.10423)
21. Ma, Y.J., Liang, W., Wang, G., Huang, D.A., Bastani, O., Jayaraman, D., Zhu, Y., Fan, L., Anandkumar, A.: Eureka: Human-level reward design via coding large language models (2023)
22. Maddigan, P., Lensen, A., Xue, B.: Explaining genetic programming trees using large language models (2024). *ArXiv* [arXiv: 2403.03397](https://arxiv.org/abs/2403.03397), <https://api.semanticscholar.org/CorpusID:268253147>
23. McDermott, J., O'Reilly, U.M.: Genetic Programming. Springer Handbook of Computational Intelligence, pp. 845–869 (2015)
24. Meyerson, E., Nelson, M.J., Bradley, H., Moradi, A., Hoover, A.K., Lehman, J.: Language model crossover: variation through few-shot prompting (2023)
25. Nasir, M.U., Earle, S., Togelius, J., James, S.D., Cleghorn, C.W.: Llmatic: neural architecture search via large language models and quality-diversity optimization (2023). *ArXiv* [arXiv: 2306.01102](https://arxiv.org/abs/2306.01102), <https://api.semanticscholar.org/CorpusID:259064000>
26. OpenAI: Gpt-4 technical report (2023)

27. O'Reilly, U.M., Toutouh, J., Pertierra, M.A., Sanchez, D.P., Garcia, D., Luogo, A.E., Kelly, J., Hemberg, E.: Adversarial genetic programming for cyber security: a rising application domain where GP matters. *Genet. Program. Evol. Mach.* **21**, 219–250 (2020). <https://api.semanticscholar.org/CorpusID:214783830>
28. Pinna, G., Ravalico, D., Rovito, L., Manzoni, L., Lorenzo, A.D.: Enhancing large language models-based code generation by leveraging genetic improvement. In: *European Conference on Genetic Programming* (2024). <https://api.semanticscholar.org/CorpusID:268999862>
29. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al.: Language models are unsupervised multitask learners. *OpenAI Blog* **1**(8), 9 (2019)
30. Raukur, T., Ho, A.C., Casper, S., Hadfield-Menell, D.: Toward transparent AI: a survey on interpreting the inner structures of deep neural networks. In: *2023 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, pp. 464–483 (2022). <https://api.semanticscholar.org/CorpusID:251104722>
31. Romera-Paredes, B., Barekatin, M., Novikov, A., Balog, M., Kumar, M.P., Dupont, E., Ruiz, F.J.R., Ellenberg, J.S., Wang, P., Fawzi, O., Kohli, P., Fawzi, A., Grochow, J., Lodi, A., Mouret, J.B., Ringer, T., Yu, T.: Mathematical discoveries from program search with large language models. *Nature* **625**, 468–475 (2023). <https://api.semanticscholar.org/CorpusID:266223700>
32. Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al.: Code llama: open foundation models for code (2023). ArXiv preprint [arXiv:2308.12950](https://arxiv.org/abs/2308.12950)
33. Sobania, D., Briesch, M., Hanna, C., Petke, J.: An analysis of the automatic bug fixing performance of ChatGPT. In: *ACM International Workshop on Automated Program Repair*, April, pp. 23–30. IEEE (2023)
34. Sobania, D., Briesch, M., Rothlauf, F.: Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'22*, pp. 1019–1027. Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3512290.3528700>
35. Song, X., Tian, Y., Lange, R.T., Lee, C., Tang, Y., Chen, Y.: Position paper: leveraging foundational models for black-box optimization: benefits, challenges, and future directions (2024). ArXiv preprint [arXiv:2405.03547](https://arxiv.org/abs/2405.03547)
36. Sudhakaran, S., González-Duque, M., Freiberger, M., Glanois, C., Najarro, E., Risi, S.: MarioGPT: Open-ended text2level generation through large language models (2024)
37. Tao, N., Ventresque, A., Saber, T.: Program synthesis with generative pre-trained transformers and grammar-guided genetic programming grammar. In: *2023 IEEE Latin American Conference on Computational Intelligence (LA-CCI)*, pp. 1–6 (2023). <https://api.semanticscholar.org/CorpusID:267260102>
38. Tao, Z., Lin, T.E., Chen, X., Li, H., Wu, Y., Li, Y., Jin, Z., Huang, F., Tao, D., Zhou, J.: A survey on self-evolution of large language models (2024). <https://api.semanticscholar.org/CorpusID:269294023>
39. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS'17*, pp. 6000–6010. Curran Associates Inc., Red Hook, NY, USA (2017)
40. Webson, A., Pavlick, E.: Do prompt-based models really understand the meaning of their prompts? In: *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2300–2344. Association for Computational Linguistics, Seattle, USA (2022). <https://doi.org/10.18653/v1/2022.naacl-main.167>, <https://aclanthology.org/2022.naacl-main.167>
41. Wu, X., hao Wu, S., Wu, J., Feng, L., Tan, K.C.: Evolutionary computation in the era of large language model: survey and roadmap (2024). ArXiv [arXiv: 2401.10034](https://arxiv.org/abs/2401.10034), <https://api.semanticscholar.org/CorpusID:267035000>
42. Xu, C., Sun, Q., Zheng, K., Geng, X., Zhao, P., Feng, J., Tao, C., Jiang, D.: WizardLM: empowering large language models to follow complex instructions (2023). ArXiv preprint [arXiv:2304.12244](https://arxiv.org/abs/2304.12244)

43. Xu, W., Qi, Y., Evans, D.: Automatically evading classifiers. In: Proceedings of the 2016 Network and Distributed Systems Symposium, vol. 10 (2016)
44. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y.: React: synergizing reasoning and acting in language models (2023)
45. Zelikman, E., Lorch, E., Mackey, L., Kalai, A.T.: Self-taught optimizer (stop): recursively self-improving code generation. In: OPT 2023: Optimization for Machine Learning (2023)

Chapter 5

Evolving Many-Model Agents with Vector and Matrix Operations in Tangled Program Graphs



Tanya Djavaheerpour, Ali Naqvi, Eddie Zhuang, and Stephen Kelly

Abstract Tangled Program Graphs (TPGs) are highly modular, hierarchical representations for genetic programming that are well-suited to multitask learning in temporal sequence prediction tasks such as control and time series forecasting. In this work, we expand the simple scalar register machines traditionally used in TPGs to include vector and matrix memory and operations. This helps TPGs evolve versatile agents that are capable of solving partially observable control and forecasting problems simultaneously. A single agent can predict actions in discrete and continuous control tasks, as well as perform generative time series prediction.

5.1 Introduction

Life presents an endless stream of unique problems that require multi-faceted solutions. We find several examples of emergent systems that generate multi-faceted problem solvers in nature and nature-inspired computation. A common theme in these biological and artificial life systems is the emergence of *many-model* solvers, in which general-purpose structural components are repeated and repurposed many times in unique, specialized contexts. We see this, for example, in the many-model structure of the neocortex (in reality and simulation), where tens of thousands of models interact and compete to organize perception, thought, and action [1, 6, 10, 22]. Evolution and plasticity of mental models are fundamental to the emergence of the many-model brains in nature. It stands to reason that artificial life systems incor-

T. Djavaheerpour · A. Naqvi · E. Zhuang · S. Kelly (✉)
Department of Computing and Software, McMaster University, Hamilton, Canada
e-mail: spkelly@mcmaster.ca

T. Djavaheerpour
e-mail: djavaheer@mcmaster.ca

A. Naqvi
e-mail: naqvial8@mcmaster.ca

E. Zhuang
e-mail: zhuane4@mcmaster.ca

porating these processes (e.g. [21, 23]) show the greatest potential to reproduce the brain’s efficiency, speed of learning new tasks, and general problem solving power.

TPGs were initially designed to solve visual reinforcement learning problems [13] and have since been extended for supervised image classification [26], time series prediction [17], radio fingerprint identification [5], large-scale parallelisation [8], and ultra-fast inference on embedded hardware through the generation of optimized standalone C code [7]. Automatic problem decomposition has been a theme in all these works, with the modular structure of TPGs lending themselves well to multitask learning problems which can benefit from the ability to discover multiple predictive models that are specialized for particular tasks, and automatically organize them into a single agent. Switching multiple models in and out of the decision control flow and runtime leads to efficient inference and helps avoid common problems associated with continual and multitask learning such as catastrophic forgetting [19] and the distraction dilemma [34].

Many recent works also explore TPGs with a focus on utilizing temporal memory [17, 18, 27, 28]. This is a natural application, since the fundamental building blocks in TPGs are Linear Genetic Programs (LGPs) [2], in which a sequence of instructions operate on input data and store intermediate values in memory registers. When registers are configured to maintain state in temporal sequence tasks, they act as a built-in temporal memory mechanism which allows agents to build and maintain a mental model of the environment on the fly. This a requirement for operation in environments that are only partially observable [3, 31].

Traditionally, register machines in TPGs have been limited to scalar memory and a minimal set of operations, e.g. $\{+, -, \times, \div, \log, \exp, \cos\}$. These constraints have led to efficient, compact, and interpretable solutions [13, 28]. However, we are increasingly interested in evolving brain-inspired computational models. To do so, we study the types of memory and operations used in computational neuroscience (e.g. [6, 32]) and evaluate how TPGs might be extended to support these data structures and operations. Indeed, systems based on Linear Genetic Programming lend themselves well to neurocomputing models because there are no constraints on the types of memory structures or operations used. Multiple systems have emerged in recent years that evolve programs using the types of scalar, vector, and matrix manipulations common in modern artificial neural networks [9, 24, 25]. In this initial expansion of TPGs, we test the utility of vector and matrix memory and operations and show how they improve the search for many-model problem solvers.

5.2 Multitask Learning

The objective in this work is to discover programs that can operate two partially observable classic control systems (Acrobot and Pendulum) [4] and also perform recursive time series forecasting on two unrelated time series; Laser [11] (a real-world recording) and Mackey-Glass [20] (a chaotic series generated from a parameterized function), Fig. 5.1. All 4 tasks can be characterized as temporal sequence

learning problems. However, each task has unique dynamics and objectives, making it exceedingly difficult to build a single agent capable of solving them all. Addressing these unique challenges simultaneously highlights the power of many-models systems.

5.2.1 Reinforcement Learning Problems

Reinforcement Learning (RL) is a type of machine learning problem in which computational agents learn through trial-and-error interaction with the problem environment over time [30]. At each timestep, the agent observes its environment through sensor inputs $\mathbf{obs}(t)$, takes an action that changes the system state, and receives a reward signal that describes the desirability of its current situation. The goal is to develop behaviors that map observations to actions such that the summed reward over all timesteps is maximized. RL is characterized by a myriad of learning challenges including partial observability of state, delayed rewards, dynamic and multitask environments, and mixed discrete and continuous-action spaces [16].

The unique challenge in this work requires an agent to operate in partially observable versions of two RL benchmarks from the classic control literature: the discrete-action Acrobot and continuous-action Pendulum, Fig. 5.1. In the Acrobot task, two links form a double pendulum with one end fixed and the other end free, both connected by an actuated joint. The state of the system at each timestep, $\mathbf{obs}(t)$, is described by 2 state variables including angles of the first and second joint, θ_1 and θ_2 . Angular velocities, $\dot{\theta}_1$ and $\dot{\theta}_2$, are not observable to the agent. The system is controlled by applying a torque of $+1$, -1 , or 0 to the joint between the links. The objective is to swing the free end of the lower link to reach a predetermined height in the fewest possible steps. Each timestep that fails to meet this target incurs a penalty of -1 , while successfully reaching the height concludes the task and grants a final reward of 0 . The episode ends after 200 timesteps. In the Pendulum task, a pole is attached at one end to a fixed point and the other end is free. The pendulum starts in a random position and the goal is to apply torque on the free end to swing it into an upright position and balance it with its center of gravity directly above the fixed point. The task ends after 300 timesteps. Both control tasks are episodic, with the agent's fitness equal to its average reward over 20 episodes with unique initial conditions. A post-training test procedure measures the agent's average reward over 100 episodes with start conditions not seen during training. Reward functions are shown in Table 5.1.

5.2.2 Recursive Time Series Forecasting Problems

The goal of time series forecasting is to predict (unseen) future values based on previously observed values. To do so, the agent is fed individual samples in order from series $x()$ and, given sample $x(t)$, must predict the value of $x(t + 1)$. Our

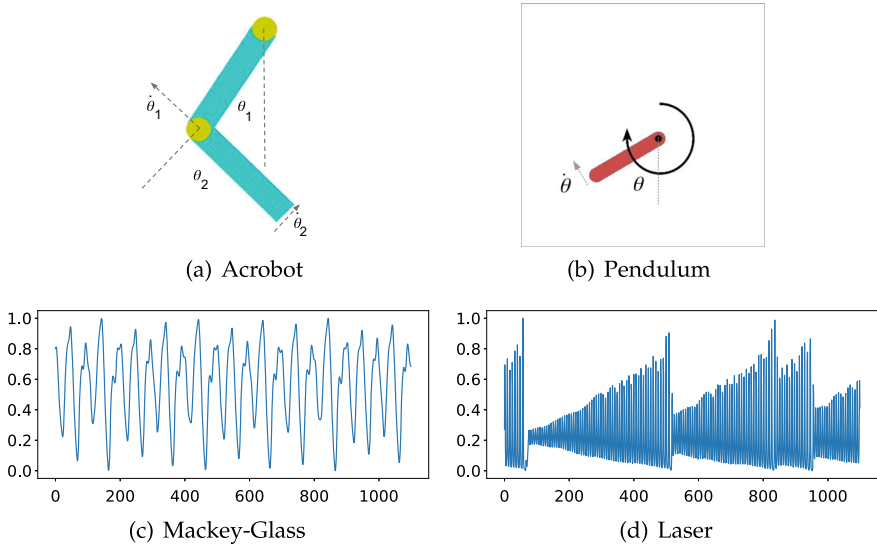


Fig. 5.1 Problems environments used in this work. For complete details on control tasks see [4] and for time series tasks see [11, 20]

Table 5.1 Definition of rewards for control tasks, provided to the agent when an episode ends due to success, failure, or a time constraint. $t_{\max}$ is the max timesteps per episode. In the Pendulum task, $\dot{\theta}$ is the angular velocity, $\text{Torque} \in (-1, 1)$ is the torque applied to the joint, and ϕ is a function to normalize the pole angle: $\phi(\theta) = ((\theta + \pi) \bmod (2 \times \pi)) - \pi$

Task	Episode reward	$t_{\max}$
Acrobot	$\sum_{t=1}^{t_{\text{end}}} -1.0$	200
Pendulum	$\sum_{t=1}^{t_{\max}} -(\phi(\theta))^2 + 0.1 \times \dot{\theta}^2 + 0.001 \times \text{Torque}^2$	300

choice of datasets and methodology was inspired by [33] and this study follows prior evaluations of TPGs in forecasting tasks [17]. The Laser and Mackey-glass datasets are univariate and contain 1100 samples normalized to the interval $[0, 1]$.

Before predictions begin at $x(t)$, the agent is executed for each input sample in series $x(t - 50)$, $x(t - 49)$, ..., $x(t - 1)$ and the agent's output values are ignored. When this *priming* phase is complete, recursive forecasting is used to predict future values. That is, after $x(t - 1)$, true samples from $x()$ are no longer used as input. Instead, the agent's predictions are fed back as input to predict future values. For example, the model's output for $x(t)$ becomes its input observation at $t + 1$, and so on. Recursive forecasting allows predictions to any horizon.

Our configuration for training evaluation uses slices of the dataset as training episodes. Beginning at t_0 , agents are *primed* with 50 samples, and the fitness function measures how well they recursively predict the next 50 samples, repeating with start points from t_{50} , t_{100} , ..., t_{950} . Thus, solution fitness is the MSE over 950 predictions

in total. In our setup, $\mathbf{obs}(t)$ stores the 32 most recent input values in a first-in-first-out buffer initialized with zeroes.

A validation procedure measures how well each agent recursively forecasts beyond the horizon used during training. Specifically, agents are used to predict the next 100 samples starting from $t_{100}, t_{200}, \dots, t_{900}$. MSE over the last 50 predictions from each validation set (a total of 450 predictions) is used as the validation score. To obtain a final test score, the single program graph with the best validation score is used to predict the next 100 samples from t_{1000} (i.e. the model is primed with samples $s(t_{950} \dots t_{999})$).

Note that $\mathbf{obs}(t)$ never contains a task-label input to identify which task is currently being experienced by the agent. As such, successful multitask agents must (1) identify the task by tracking how observation variables change over time and (2) predict the best task-specific action/output at each timestep.

5.2.3 Multitask Fitness

During evolution, each agent in the population is evaluated on each task independently, resulting in 4 task-specific *raw fitness* scores. Agents are then assigned a *multitask* fitness for each set $s \in \mathbb{P}$ (the power set of 4 task combinations). For single-task sets, multitask fitness is simply the training fitness in that task. For multitask sets, fitness captures how well an agent performs on multiple problems by ranking each agent by their weakest performance in the problem set. To achieve this, every agent's mean reward on each task is normalized relative to the rest of the current population. Normalized score for agent a_i on task t_j is calculated as follows:

$$sc^{\text{nm}}(a_i, t_j) = (sc(a_i, t_j) - sc_{\min}(t_j)) / (sc_{\max}(t_j) - sc_{\min}(t_j)) \quad (5.1)$$

where $sc(a_i, t_j)$ is the mean score for agent a_i on task t_j and $sc_{\min, \max}(t_j)$ are the population-wide min and max mean scores for task t_j . Fitness for agent a_i is then $\min(sc^{\text{nm}}(a_i, t_{\{1..n\}}))$, or the minimum normalized score for agent a_i over all problems. n denotes the number of problems. Thus, *champions* are the agents with the highest minimum normalized fitness over all problems in each set. Normalizing rewards is a critical part of quantifying multitask fitness and mitigates the *distraction dilemma* [34], a common issue in multitask learning in which the differing magnitude of reward signals may make certain tasks appear more salient.

5.3 Tangled Program Graphs

Tangled Program Graphs (TPGs) are modular, hierarchical representations for Genetic Programming that are particularly well-suited to multitask learning in partially observable visual reinforcement learning and time series prediction tasks [17].

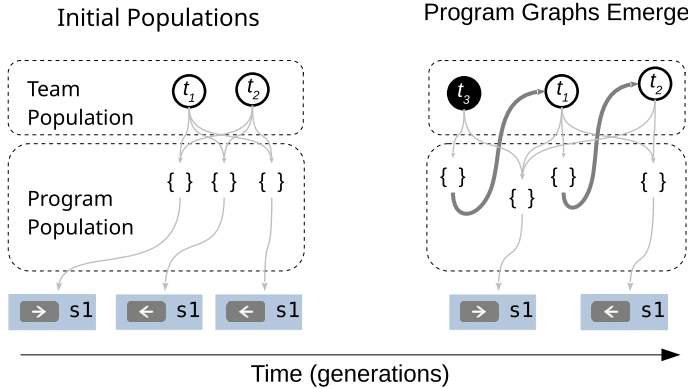


Fig. 5.2 Illustration of the relationship between teams and programs in TPG. Initially, all programs are leaf nodes. Over time, program action pointers may be modified to refer to other teams and *program graphs* emerge. When a team is subsumed into a program graph, it is cloned and the clone (t_{2c}) becomes an internal node. See Sect. 5.3.3 for details

In short, TPGs excel at tasks which require automatic problem decomposition and temporal memory. The big idea is to construct many-model prediction machines from the bottom up, adaptively building programs, teams of programs, and graphs of teams of programs through an emergent and open-ended evolutionary search, Fig. 5.2. The following subsections describe each component in detail and outline the fundamental rules that govern their interaction with each other and the external environment.

5.3.1 Programs

The foundational representation in TPGs are linear genetic programs, or register machines [2]. Register machines execute a sequence of operations which transform data stored in memory, Algorithm 5.1. Register memory may be organized in virtually any data structure, where the types of data structures used will define the set of operations a program may use to manipulate that data. Inspired by AutoML-Zero [25], this work supports scalar, vector, and matrix memory structures, and a wide array of potential operations, Table 5.3. Three properties of the register machine representation are particularly important for TPGs:

1. Since programs operate on memory structures, all intermediate values are stored in memory, and are therefore accessible to any other instruction in the program. In temporal sequence tasks, memory structures can maintain state between each step in the sequence. This *stateful* property allows the agent to store and integrate data over time, forming the basis of a dynamic *mental model*.

2. Any individual memory structure, or set of structures, may be interpreted as the program's *output* post-execution. As such, single or multiple outputs may be defined, and outputs may take the form of any data structures supported by the machine.
3. Instructions that have no effect on the value of output registers can be identified prior to execution and skipped during evaluation of the agent. A significant proportion of instructions may be ineffective in this respect. Given that each evaluation in temporal sequence tasks may require hundreds of program executions, skipping these *intron* instructions can have a large impact on the speed and efficiency of agents.

Algorithm 5.1 Example program (register machine). Each program contains 8 instances of each memory type, scalar s , vector v , and matrix m . All integers in the program (e.g. numbers following s, v, m) are evolved constants used for memory indexing. m_w is an evolved integer constant, unique to each program, which specifies the dimensionality of vector and matrix memory, sized $(m_w, 1)$ and (m_w, m_w) respectively. o_i is an evolved integer constant specifying this program's observation offset index. $\mathbf{obs}(t)$ represents the observable state at time t . Python code in lines 1–3 copy $\mathbf{obs}(t)$ to $v0$ and $m0$, making it accessible to vector and matrix operations later in the program (See Fig. 5.3 for an example). Scalar operations can access $\mathbf{obs}(t)$ directly (e.g. line 7). All memories are stateful, meaning their contents are reset to zero at the beginning of each agent evaluation in a given task, and thereafter left to accumulate intermediate values over time. Programs have two return values (line 11) which are interpreted as the weight (confidence) value and action output. In this example line 8 does not affect the final value of $s0$ or $s1$. Ineffective instructions are useful for the evolutionary search [35], but for efficiency they can easily be identified and skipped during program execution [2]. A complete list of operations and instruction formats appears in Table 5.3

```

1: v0 = numpy.roll(obs(t), -oi)[:mw]           ▷ Copy observation to vector memory
2: v1 = numpy.roll(obs(t), -oi)[mw*mw]       ▷ Copy observation to temporary vector v1
3: m0 = v1.reshape(mw, mw)                     ▷ Copy observation to matrix memory
4: v3 = s0*v1                                     ▷ Program execution begins
5: v3 = s0*v1
6: s0 = mean(v3)
7: s0 = s0/v1[3]
8: s3 = norm(m1)                                  ▷ Intron
9: s1 = cos(s0)
10: if (s0 < v1[2]): s0 = -s0
11: return s0, s1                                ▷ weight, continuous action

```

Programs represent the smallest distinct instance of a predictive *model*, Algorithm 1. In this work, they have two return values which are interpreted in unique ways. Scalar $s0$ can be characterized as a *confidence* value which predicts the suitability of this program's other return values relative to the current state of the system. Programs also return scalar $s1$, which can be used as a continuous-valued *action* in control tasks,

		0	1	2	3	4	5	6	7
obs(t)	→	0.1	1.3	2.0	0.3	5.8	1.0	0.0	6.1
v0	→	0.0	6.1	0.1					
m0	→	0.0	6.1	0.1	1.3	2.0	0.3	5.8	1.0

Fig. 5.3 Illustration of how vector observation data **obs(t)** is copied to vector and matrix program memory, lines 1–3 in Algorithm 5.1. In this example, **obs(t)** is size 8, the program’s evolved memory dimensionality m_w is 3, and its evolved observation offset index o_i is 6. This methodology allows for observation data to be represented as any size vector or matrix. Since programs have their own mutable memory dimensionality and offset, they are free to access any part of the observation through a *window* of this size

or a prediction variable in time series forecasting. Finally, to support discrete control tasks, programs are also associated with one task-specific discrete-action pointer, See Fig. 5.2. In short, the role of programs is to learn the appropriate *contexts* in which their continuous output register ($s1$) and/or discrete action should be deployed to the environment (Fig. 5.3).

5.3.2 Teams of Programs

In order to explicitly encourage problem decomposition, individual programs do not work alone. Instead, a *team of programs* is the basic representation for a standalone agent in TPGs. Each team is represented as a list of programs that *collectively* map input observations **obs(t)** to a pair of discrete and continuous actions. Teams can be thought of as vertices in a computational graph where the programs represent directed, dynamically weighted *edges*. For example (see Fig. 5.2), given observation **obs(t)**, all programs in team t_1 will be executed in order. The contents of each program’s $s0$ are interpreted as its weight. The edge with the largest weight defines the agent’s output at time t . In this work, we require a discrete action (for discrete control tasks such as Acrobot) and a single continuous value (for continuous control tasks such as Pendulum and time series prediction). As such the atomic (i.e. terminal) program returns its discrete-action pointer and the value stored in $s1$. Note that any

type of memory may be used for output, implying this framework is extensible to various task types (e.g. returning a vector memory instead of a scalar would allow this representation to be used in multivariate time series prediction or robot control requiring multiple continuous control variables [15].)

5.3.3 *Graphs of Teams of Programs*

Programs and Teams are stored in separate populations and coevolved, Fig. 5.2. Initially, each team is composed of 3 unique programs and all programs are leaf nodes, as in the left-hand side of Fig. 5.2. Evolution is driven by a generational Genetic Algorithm (GA) in which parent selection identifies the most promising root teams,¹ which are then modified by crossover and mutation operators.

Team variation operators may change the program order or add, remove, and mutate programs in the team. Team complement as well as program length and content are all adapted properties. When a program is modified by variation operators, it will remain a leaf with probability p_{atomic} and will otherwise connect to one team from the set of teams present from any previous generation. These connection mutations are the mechanism by which TPGs support compositional evolution, adaptively recombining multiple (previously independent) teams into graph structures, or *program graphs*, Right-hand side of Fig. 5.2.²

Execution of a program graph begins at the root team (t_3 in Fig. 5.2), where all programs in the team will execute in order. Graph traversal then follows the program with the largest weight, repeating the execution process at every team along the path until a leaf node is reached. Thus, the graph computes one path from root to leaf at each timestep, where only a subset of programs in the graph (those in teams along the path) require execution. This process is illustrated in Algorithm 5.2.

The hierarchical inter-dependency between teams emerges entirely through interaction with the task environment. The subset of teams/programs that require execution is dynamically selected at runtime as a function of the agent's input observation and the state of its mental model (i.e. memory). This many-model representation allows evolution to build agents for multiple problems simultaneously [18].

¹ During selection, fitness ties are broken by comparing the inference complexity of agents, with lower complexity preferred. This helps control bloat by reducing the likelihood of agents complexifying without any associated fitness gain (see Sect. 5.4.2).

² Complete details are available in [12, 13, 18].

Algorithm 5.2 Selecting an action through traversal of a program graph. P is the current program population. tm_i is the current team (initially a root node). $\mathbf{obs}(t)$ is the state observation. V is the set of teams visited throughout *this* traversal (initially empty). First, all programs in tm_i are executed relative to the current state $\mathbf{obs}(t)$ (Lines 4, 5). The algorithm then considers each program in order of weight (highest to lowest, Line 8). If the program is a terminal node, its discrete and continuous action pair is returned (Line 9). Otherwise, if the program’s action points to a team that has not yet been visited, the procedure is called recursively on that team. Thus, while a program graph may contain cycles, they are not followed during traversal. To ensure an atomic is always found, team variation operators are constrained such that each team maintains at least one program that has an atomic action

```

1: procedure SelectAction( $tm_i, \mathbf{obs}(t), V$ )
2:    $A = \{x \in P : x \text{ has atomic action}\}$ 
3:    $V = V \cup tm_i$  ▷ add  $tm_i$  to visited teams
4:   for all  $p_i \in tm_i$  do
5:      $weight(p_i) = exec(p_i, \mathbf{obs}(t))$  ▷ run prog. on  $\mathbf{obs}(t)$  and save result
6:   end for
7:    $tm'_i = sort(tm_i)$  ▷ sort progs by weight, highest to lowest
8:   for all  $p_i \in tm'_i$  do
9:     if  $p_i \in A$  then return  $action(p_i)$  ▷ atomic reached
10:    else if  $action(p_i) \notin V$  then
11:      return  $SelectAction(action(p_i), \mathbf{obs}(t), V)$  ▷ delegate action (graph edge)
12:    end if
13:  end for
14: end procedure

```

5.4 Experimental Results

Experiments in this work are primarily designed to benchmark the utility of vector and matrix operations in TPGs. Results suggest that these extensions can significantly improve multitask sequence learning in the challenging 4-task scenario described in Sect. 5.2. This section discusses these findings in terms of problem solving ability and inference complexity of the resulting multitask agents.

5.4.1 Problem Solving Ability

Figure 5.4 reports test fitness at each generation for multitask champions interacting with each problem. Two independent experimental cases are shown, scalar-only TPGs and TPGs extended with vector and matrix operations. We run 50 repeats of each experiment, with hyperparameters listed in Table 5.2. In the Mackey-Glass and Laser tasks, TPGs with vector and matrix operations are consistently outperforming those without. Figure 5.4 shows the median fitness (solid line) and min/max fitness

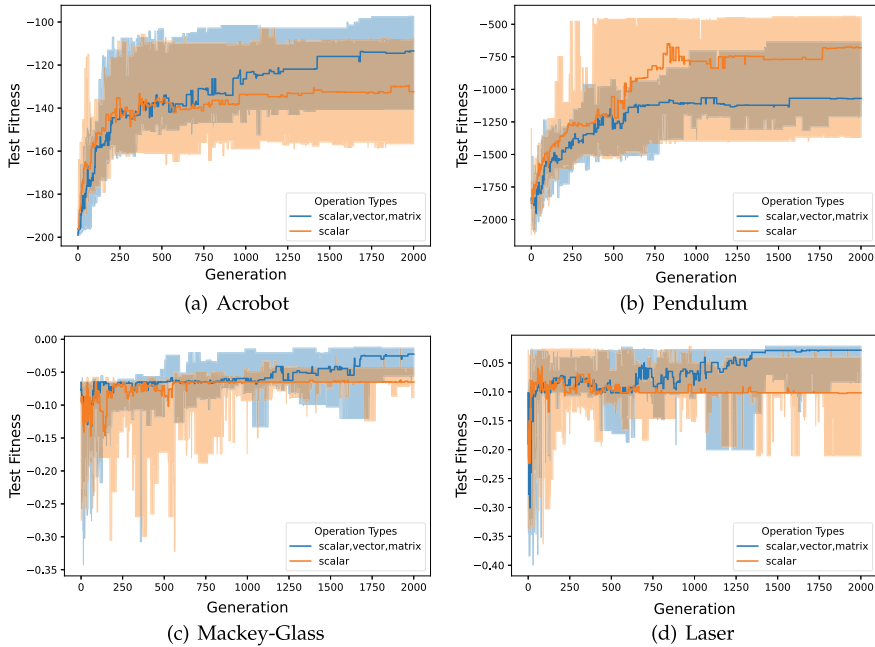


Fig. 5.4 Evolutionary progress in the multitask learning challenge. Plot shows the median test fitness (solid line) and min/max fitness (shaded area) for the single best multitask agent over 50 repeats. The fitness score at each generation is produced by the *same* multitask agent which solves all 4 problems relatively well. This agent is selected from the population using the multitask ranking metric described in Sect. 5.2.3

(shaded area) for the single best multitask agent in each repeat. The Mann-Whitney U test is applied to compare the final test fitness for the pair of experiments in each plot. The *scalar, vector, matrix* case is significantly better in Mackey-Glass and Laser ($p < 0.05$), while the *scalar* case is better in Pendulum ($p < 0.05$), and no significant difference is found in Acrobot. Overall, only the *scalar, vector, matrix* case is able to discover a single multitask agent capable of solving all 4 tasks. That is, the agent can swing up the Acrobot, swing up and balance the Pendulum, and achieves test MSE in the recursive forecasting tasks which matches the best results achieved by single-task agents in previous work³ [14]. This agent's prediction accuracy in recursive forecasting is shown in Fig. 5.5.

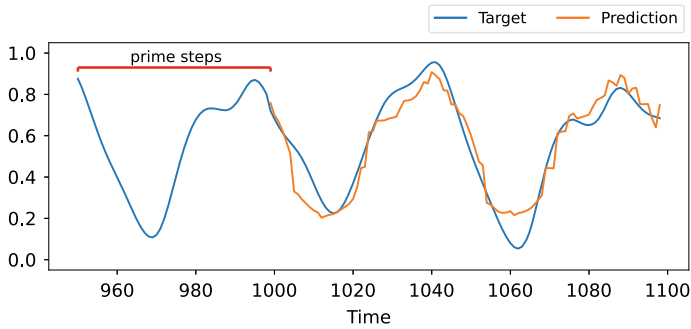
³ The best *scalar, vector, matrix* agent in Fig. 5.4 achieve a test score of 0.0144 and 0.0127 for Mackey-Glass and Laser, respectively.

Table 5.2 Parameterization of team and program populations. n_{elite} is the number of agents to save in each generation. For the team population, p_x is the probability of crossover and p_{mx} denotes a mutation operator in which: $x \in \{d, a\}$ are the probability of deleting or adding a program, respectively; $x \in \{m, n, s\}$ are the probability of creating a new program, changing a program's action pointer (leaf or team), and changing a program's shared memory pointer, respectively. For the program population, p_x denotes a mutation operator in which $x \in \{\text{delete, add, mutate, swap}\}$ are the probability for deleting, adding, mutating, or reordering instructions within a program. m_w is the dimensionality of a program's vector and matrix memory. p_{mem} is the probability of changing m_w . p_{atomic} is the probability that a modified action pointer will be atomic (terminal)

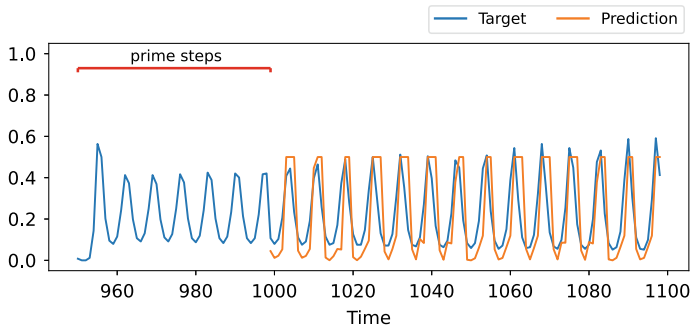
Team population			
Parameter	Value	Parameter	Value
Agent (root team) population size	1000	p_{md}	0.5
n_{elite}	500	p_{ma}	0.4
Initial team size	10	p_{mn}, p_{ms}	0.1
Max team size	∞	p_{mm}	0.2
		p_x	0.5
Program population			
Parameter	Value	Parameter	Value
Number of elements in $\mathbf{s}$	8	Initial m_w	2
Number of elements in $\mathbf{m}$	8	Min m_w	2
Number of elements in $\mathbf{v}$	8	Max m_w	8
Initial program size	10	Max program size	∞
p_{delete}	0.5	p_{add}	0.4
p_{mutate}	1.0	p_{swap}	0.2
p_{atomic}	0.99	p_{mem}	0.2

5.4.2 Inference Complexity

Figure 5.6 provides the average number of program instructions executed per prediction by the best agent as it interacts with each task. *Complexification* [29] is clearly visible in most cases. The agents start simple and gradually complexify through interaction with the problems. The only significant difference found between experimental cases is in the recursive forecasting tasks. In the scalar-only experiments, fitness tends to stagnate at generation ≈ 250 . It seems that evolution does not discover ways to improve simple agents, and incremental code growth does not result in consistent fitness improvements. The population tends not to complexify as a result. On the other hand, the *scalar*, *vector*, *matrix* cases exhibit incremental growth and steady fitness improvements beyond generation 1000 in all tasks except Pendulum. It appears that the extended memory and operation set enhances the expressive power of TPGs in the recursive forecasting tasks. We leave it to future work to analyze which specific operations are important and to interpret how they are used within linear genetic programs.



(a) Mackey-Glass



(b) Laser

Fig. 5.5 Example recursive forecasts produced by the best multitask agent with the full set of operations (See *scalar*, *vector*, *matrix* in Fig. 5.4)

5.4.3 Evolved Many-Model Hierarchies

Figure 5.7 depicts an example program graph in which each node represents one team of programs. Node charts illustrate the proportion of timesteps in which each team was visited during test episodes in each task. For example, the root node is visited in every timestep, thus proportions are equal for **Acrobot**, **Pendulum**, **Mackey-Glass**, and **Laser**. It is currently unclear why long vertical paths appear traversing several teams with equivalent task distributions. This structure implies that all teams in such paths are visited together, and internal nodes are potentially redundant. In the current version of TPGs, variation only operates on root nodes, thus program graphs can only be constructed from the bottom up, and internal nodes are essentially frozen, protected from variation. This property is helpful in avoiding catastrophic forgetting in dynamic and multitask environments that require continual learning, but it comes with reduced ability for the search to fine tune behaviors by adjusting internal components of the agent's structure. Given that the example team in Fig. 5.7 is competent in all tasks, we wonder if support for internal node deletion

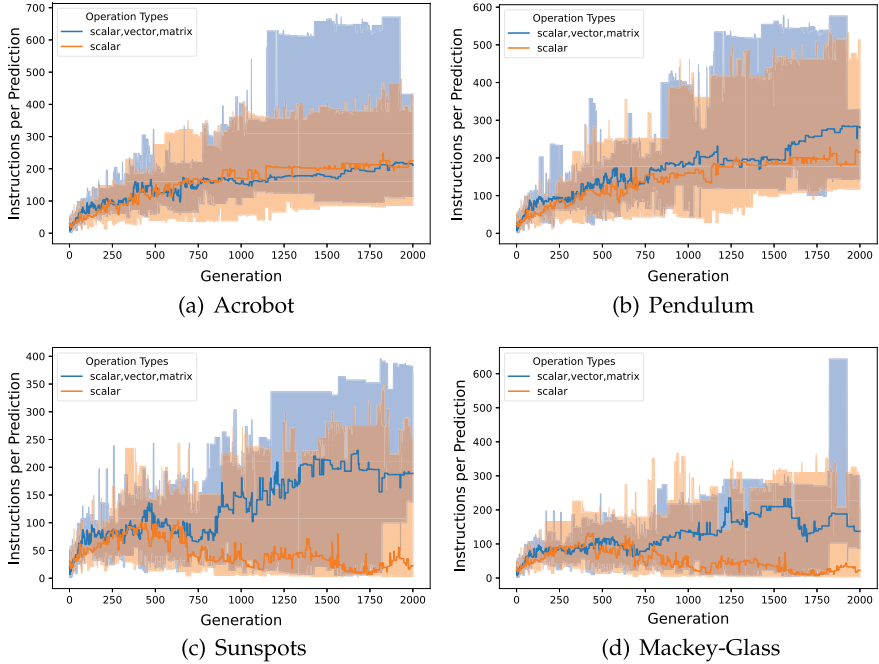


Fig. 5.6 Development of agent complexity. Plot shows the average number of program instructions executed per prediction by the best multitask agent as it interacts with each task. Median (solid line) and min/max (shaded area) over 50 independent repeats are shown. The Mann-Whitney U test is applied to compare final instruction counts for the pair of experiments in each plot. Significant difference is found in Mackey-Glass and Laser ($p < 0.05$). See Fig. 5.3 for an illustrative example program

and/or variation could reduce the structural complexity of graphs without having to *tear down* the agent from the root and build it back up. Answering this question presents an important opportunity for future work.

5.5 Discussion and Future Work

TPGs evolve many-model agents which reuse structurally simple building blocks in complex hierarchical systems. These agents are evolved from scratch to solve several types of temporal sequence prediction problems simultaneously. We have shown that TPGs can discover multi-faceted solvers for a set partially observable problems including discrete control, continuous control, and two unrelated time series prediction tasks. For the first time, we have demonstrated the utility of vector and matrix operations in TPGs. These extensions are most useful in the forecasting tasks, which have larger observation spaces (control tasks have $\mathbf{obs}(t)$ of size 2,

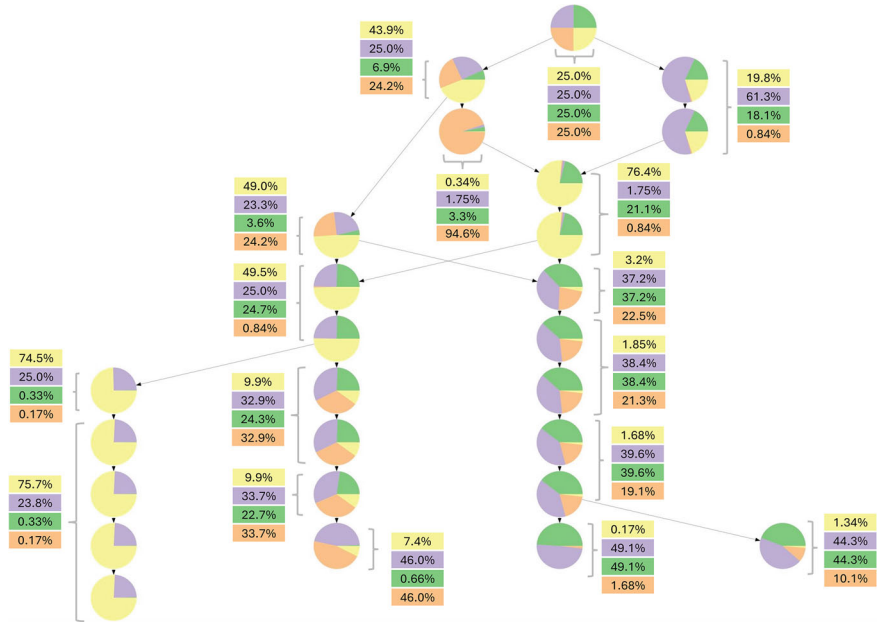


Fig. 5.7 Example program graph in which each node represents one team of programs. Node charts illustrate the proportion of timesteps in which each team was visited during test episodes in each task

while recursive forecasting tasks have $\mathbf{obs}(t)$ of size 32). Furthermore, unlike the control tasks, forecasting tasks encode a time window in $\mathbf{obs}(t)$. These differences are likely the main reason why vector and matrix memory and linear algebra operations provide an advantage over the simple scalar register machines traditionally used in TPGs. However, this study is just the beginning. Future work will analyze which specific memories and operations are important and interpret how they are used within TPGs. We are also interested modifying TPGs' graph-building operations in order to optimize many-model agents for efficiency. Broadly speaking, the long-term goal is to design a search space which includes the operations needed to discover new algorithms inspired by advanced neurocomputing models such as Hierarchical Temporal Memory (HTM) [6] and to define new operators which can efficiently guide evolution in this direction.

Table 5.3 Set of operations available to programs. s , $\mathbf{v}$, and M denote a scalar, vector, and matrix, respectively. Early-alphabet letters (a , b , etc.) denote memory addresses. Mid-alphabet letters (e.g. i , j , etc.) denote indexes. In this work, the *scalar* experimental case is limited to OP0 – OP12, while all operations are available in the *scalar*, *vector*, *matrix* case. Operations are inspired by AutoML-Zero [25]

Op ID	Code example	Input args addresses/types	Output args address	Description (see caption)/types
OP0	<code>s2=s3+s0</code>	$a, b/\text{scalars}$	c/scalar	$s_c = s_a + s_b$
OP1	<code>s4=s0-s1</code>	$a, b/\text{scalars}$	c/scalar	$s_c = s_a - s_b$
OP2	<code>s8=s5*s5</code>	$a, b/\text{scalars}$	c/scalar	$s_c = s_a s_b$
OP3	<code>s7=s5/s2</code>	$a, b/\text{scalars}$	c/scalar	$s_c = s_a / s_b$
OP4	<code>s8=abs(s0)</code>	a/scalar	b/scalar	$s_b = s_a $
OP5	<code>s4=1/s8</code>	a/scalar	b/scalar	$s_b = 1/s_a$
OP6	<code>s1=cos(s4)</code>	a/scalar	b/scalar	$s_b = \cos(s_a)$
OP7	<code>s1=exp(s2)</code>	a/scalar	b/scalar	$s_b = e^{s_a}$
OP8	<code>s0=log(s3)</code>	a/scalar	b/scalar	$s_b = \log s_a$
OP9	<code>s3=heaviside(s0)</code>	a/scalar	b/scalar	$s_b = \mathbb{1}_{\mathbb{R}^+}(s_a)$
OP10	<code>if(s3<s7):s2=-s2</code>	$a, b/\text{scalar}$	c/scalar	IF($s_a < s_b$) THEN $s_c = -s_c$
OP11	<code>s1=minimum(s2,s3)</code>	$a, b/\text{scalars}$	c/scalar	$s_c = \min(s_a, s_b)$
OP12	<code>s8=maximum(s3,s0)</code>	$a, b/\text{scalars}$	c/scalar	$s_c = \max(s_a, s_b)$
OP13	<code>v2=heaviside(v2)</code>	a/vector	b/vector	$\mathbf{v}_b^{(i)} = \mathbb{1}_{\mathbb{R}^+}(\mathbf{v}_a^{(i)}) \forall i$
OP14	<code>m7=heaviside(m3)</code>	a/matrix	b/matrix	$M_b^{(i,j)} = \mathbb{1}_{\mathbb{R}^+}(M_a^{(i,j)}) \forall i, j$
OP15	<code>v1=s7*v1</code>	$a, b/\text{sc,vec}$	c/vector	$\mathbf{v}_c = s_a \mathbf{v}_b$
OP16	<code>v1=bcast(s3)</code>	a/scalar	b/vector	$\mathbf{v}_b^{(i)} = s_a \forall i$
OP17	<code>v5=1/v7</code>	a/vector	b/vector	$\mathbf{v}_b^{(i)} = 1/\mathbf{v}_a^{(i)} \forall i$
OP18	<code>s0=norm(v3)</code>	a/scalar	b/vector	$s_b = \mathbf{v}_a $
OP19	<code>v3=abs(v3)</code>	a/vector	b/vector	$\mathbf{v}_b^{(i)} = \mathbf{v}_a^{(i)} \forall i$
OP20	<code>v5=v0+v9</code>	$a, b/\text{vectors}$	c/vector	$\mathbf{v}_c = \mathbf{v}_a + \mathbf{v}_b$
OP21	<code>v1=v0-v9</code>	$a, b/\text{vectors}$	c/vector	$\mathbf{v}_c = \mathbf{v}_a - \mathbf{v}_b$
OP21	<code>v8=v1*v9</code>	$a, b/\text{vectors}$	c/vector	$\mathbf{v}_c^{(i)} = \mathbf{v}_a^{(i)} \mathbf{v}_b^{(i)} \forall i$
OP22	<code>v9=v8/v2</code>	$a, b/\text{vectors}$	c/vector	$\mathbf{v}_c^{(i)} = \mathbf{v}_a^{(i)} / \mathbf{v}_b^{(i)} \forall i$
OP23	<code>s6=dot(v1,v5)</code>	$a, b/\text{vectors}$	c/scalar	$s_c = \mathbf{v}_a^T \mathbf{v}_b$
OP24	<code>v4=minimum(v3,v9)</code>	$a, b/\text{vectors}$	c/vector	$\mathbf{v}_c^{(i)} = \min(\mathbf{v}_a^{(i)}, \mathbf{v}_b^{(i)}) \forall i$
OP25	<code>v7=maximum(v3,v6)</code>	$a, b/\text{vectors}$	c/vector	$\mathbf{v}_c^{(i)} = \max(\mathbf{v}_a^{(i)}, \mathbf{v}_b^{(i)}) \forall i$
OP26	<code>s2=mean(v2)</code>	a/vector	b/scalar	$s_b = \text{mean}(\mathbf{v}_a)$
OP27	<code>s3=std(v3)</code>	a/vector	b/scalar	$s_b = \text{stdev}(\mathbf{v}_a)$
OP28	<code>m1=outer(v6,v5)</code>	$a, b/\text{vectors}$	c/matrix	$M_c = \mathbf{v}_a \mathbf{v}_b^T$
OP29	<code>m1=s4*m2</code>	$a, b/\text{sc/mat}$	c/matrix	$M_c = s_a M_b$
OP30	<code>m3=1/m0</code>	a/matrix	b/matrix	$M_b^{(i,j)} = 1/M_a^{(i,j)} \forall i, j$
OP31	<code>v6=dot(m1,v0)</code>	$a, b/\text{mat/vec}$	c/vector	$\mathbf{v}_c = M_a \mathbf{v}_b$
OP32	<code>m2=bcast(v0,axis=0)</code>	a/vector	b/matrix	$M_b^{(i,j)} = \mathbf{v}_a^{(i)} \forall i, j$
OP33	<code>m2=bcast(v0,axis=1)</code>	a/vector	b/matrix	$M_b^{(j,i)} = \mathbf{v}_a^{(i)} \forall i, j$
OP34	<code>s2=norm(m1)</code>	a/matrix	b/scalar	$s_b = M_a $
OP35	<code>v4=norm(m7,axis=0)</code>	a/matrix	b/vector	$\mathbf{v}_b^{(i)} = M_a^{(i,\cdot)} \forall i$

(continued)

Table 5.3 (continued)

Op ID	Code example	Input args addresses/types	Output args address	Description (see caption)/types
OP36	<code>v4=norm(m7,axis=1)</code>	<i>a</i> /matrix	<i>b</i> /vector	$\mathbf{v}_b^{(j)} = \mathbf{M}_a^{(\cdot,j)} \forall j$
OP37	<code>m9=transpose(m3)</code>	<i>a</i> /matrix	<i>b</i> /matrix	$\mathbf{M}_b = \mathbf{M}_a^T $
OP38	<code>m1=abs(m8)</code>	<i>a</i> /matrix	$-b$ /matrix	$\mathbf{M}_b^{(i,j)} = \mathbf{M}_a^{(i,j)} \forall i, j$
OP39	<code>m2=m2+m0</code>	<i>a, b</i> /matrixes	<i>c</i> /matrix	$\mathbf{M}_c = \mathbf{M}_a + \mathbf{M}_b$
OP40	<code>m2=m3-m1</code>	<i>a, b</i> /matrixes	<i>c</i> /matrix	$\mathbf{M}_c = \mathbf{M}_a - \mathbf{M}_b$
OP41	<code>m3=m2*m3</code>	<i>a, b</i> /matrixes	<i>c</i> /matrix	$\mathbf{M}_c^{(i,j)} = \mathbf{M}_a^{(i,j)} \mathbf{M}_b^{(i,j)} \forall i, j$
OP42	<code>m4=m2/m4</code>	<i>a, b</i> /matrixes	<i>c</i> /matrix	$\mathbf{M}_c^{(i,j)} = \mathbf{M}_a^{(i,j)} / \mathbf{M}_b^{(i,j)} \forall i, j$
OP43	<code>m5=matmul(m5,m7)</code>	<i>a, b</i> /matrixes	<i>c</i> /matrix	$\mathbf{M}_c = \mathbf{M}_a \mathbf{M}_b$
OP44	<code>m2=minimum(m2,m1)</code>	<i>a, b</i> /matrixes	<i>c</i> /matrix	$\mathbf{M}_c^{(i,j)} = \min(\mathbf{M}_a^{(i,j)}, \mathbf{M}_b^{(i,j)}) \forall i, j$
OP45	<code>m7=maximum(m1,m0)</code>	<i>a, b</i> /matrixes	<i>c</i> /matrix	$\mathbf{M}_c^{(i,j)} = \max(\mathbf{M}_a^{(i,j)}, \mathbf{M}_b^{(i,j)}) \forall i, j$
OP46	<code>s2=mean(m8)</code>	<i>a</i> /matrix	<i>b</i> /scalar	$s_b = \text{mean}(\mathbf{M}_a)$
OP47	<code>v1=mean(m2,axis=0)</code>	<i>a</i> /matrix	<i>b</i> /vector	$\mathbf{v}_b^{(i)} = \text{mean}(\mathbf{M}_a^{(i,\cdot)}) \forall i$
OP48	<code>v3=std(m2,axis=0)</code>	<i>a</i> /matrix	<i>b</i> /vector	$\mathbf{v}_b^{(i)} = \text{stdev}(\mathbf{M}_a^{(i,\cdot)}) \forall i$
OP49	<code>s4=std(m0)</code>	<i>a</i> /matrix	<i>b</i> /scalar	$s_b = \text{stdev}(\mathbf{M}_a)$

Acknowledgements This research was enabled in part by support provided by the Digital Research Alliance of Canada (alliancecan.ca) and the NSERC Discovery Grants program.

References

1. Badcock, P.B., Friston, K.J., Ramstead, M.J.D., Ploeger, A., Hohwy, J.: The hierarchically mechanistic mind: an evolutionary systems theory of the human brain, cognition, and behavior. *Cogn. Affect. Behav. Neurosci.* **19**(6), 1319–1351 (2019)
2. Brameier, M., Banzhaf, W.: *Linear Genetic Programming*. Springer (2007)
3. Brave, S.: The evolution of memory and mental models using genetic programming. In: *Proceedings of the 1st Annual Conference on Genetic Programming*, pp. 261–266. MIT Press (1996)
4. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: *OpenAI Gym* (2016). [ArXiv: arXiv:1606.01540v1](https://arxiv.org/abs/1606.01540v1)
5. Chillet, A., Boyer, B., Gerzaguet, R., Desnos, K., Gautier, M.: Tangled program graph for radio-frequency fingerprint identification. In: *2023 Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (IEEE PIMRC 2023)* (2023)
6. Cui, Y., Ahmad, S., Hawkins, J.: Continuous online sequence learning with an unsupervised neural network model. *Neural Comput.* **28**(11), 2474–2504 (2016)
7. Desnos, K., Bourgoin, T., Dardaillon, M., Sourbier, N., Gesny, O., Pelcat, M.: Ultra-fast machine learning inference through C code generation for tangled program graphs. In: *2022 IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 1–6 (2022)
8. Desnos, K., Sourbier, N., Raumer, P.Y., Gesny, O., Pelcat, M.: Gegalati: lightweight artificial intelligence through generic and evolvable tangled program Graphs. In: *Workshop on Design*

- and Architectures for Signal and Image Processing. DASIP'21, 14th ed., pp. 35–43. ACM, New York, NY, USA (2021)
9. Gligorovski, N., Zhong, J.: Lgp-vec: A vectorial linear genetic programming for symbolic regression. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation, GECCO'23 Companion, p. 579–582. Association for Computing Machinery, New York, NY, USA (2023)
 10. Hawkins, J., Ahmad, S., Cui, Y.: A theory of how columns in the neocortex enable learning the structure of the world. *Front. Neural Circ.* **11**, 81 (2017)
 11. Hübner, U., Abraham, N.B., Weiss, C.O.: Dimensions and entropies of chaotic intensity pulsations in a single-mode far-infrared nh_3 laser. *Phys. Rev. A* **40**, 6354–6365 (1989)
 12. Kelly, S.: Scaling genetic programming to challenging reinforcement tasks through emergent modularity. Ph.D. thesis, Faculty of Computer Science, Dalhousie University (2018)
 13. Kelly, S., Heywood, M.I.: Emergent solutions to high-dimensional multitask reinforcement learning. *Evol. Comput.* **26**(3), 347–380 (2018)
 14. Kelly, S., Newsted, J., Banzhaf, W., Gondro, C.: A modular memory framework for time series prediction. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference, GECCO'20, pp. 949–957. ACM, New York, NY, USA (2020)
 15. Kelly, S., Park, D.S., Song, X., McIntire, M., Nashikkar, P., Guha, R., Banzhaf, W., Deb, K., Boddeti, V.N., Tan, J., Real, E.: Discovering adaptable symbolic algorithms from scratch. In: International Conference on Intelligent Robots and Systems (IROS) (2023)
 16. Kelly, S., Schossau, J.: Evolutionary Computation and the Reinforcement Learning Problem, pp. 79–118. Springer Nature, Singapore (2024)
 17. Kelly, S., Smith, R.J., Heywood, M.I., Banzhaf, W.: Emergent tangled program graphs in partially observable recursive forecasting and vizdoom navigation tasks. *ACM Trans. Evol. Learn. Optim.* **1**(3) (2021)
 18. Kelly, S., Voegerl, T., Banzhaf, W., Gondro, C.: Evolving hierarchical memory-prediction machines in multi-task reinforcement learning. *Genet. Program. Evol. Mach.* **22**(4), 573–605 (2021)
 19. Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A.A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., Hassabis, D., Clopath, C., Kumaran, D., Hadsell, R.: Overcoming catastrophic forgetting in neural networks. *Proc. Natl. Acad. Sci.* **114**(13), 3521–3526 (2017)
 20. Mackey, M.C., Glass, L.: Oscillation and chaos in physiological control systems. *Science* **197**(4300), 287–289 (1977)
 21. Miller, J.F.: Evolving developmental neural networks to solve multiple problems. In: ALIFE 2020: The 2020 Conference on Artificial Life, Artificial Life Conference Proceedings, pp. 473–482. ASME (2020)
 22. Mountcastle, V.B.: The columnar organization of the neocortex. *Brain* **120**, 701–722 (1997)
 23. Najarro, E., Risi, S.: Meta-learning through Hebbian plasticity in random networks. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) *Advances in Neural Information Processing Systems*, vol. 33, pp. 20,719–20,731. Curran Associates, Inc. (2020)
 24. Praczyk, T., Szymkowiak, M.: Linear matrix genetic programming as a tool for data-driven black-box control-oriented modeling in conditions of limited access to training data. *Sci. Rep.* **14**(1), 12,666 (2024)
 25. Real, E., Liang, C., So, D.R., Le, Q.V.: AutoML-zero: evolving machine learning algorithms from scratch (2020). arXiv preprint [arXiv:2003.03384](https://arxiv.org/abs/2003.03384)
 26. Smith, R.J., Amaral, R., Heywood, M.I.: Evolving simple solutions to the cifar-10 benchmark using tangled program graphs. In: 2021 IEEE Congress on Evolutionary Computation (CEC), pp. 2061–2068 (2021)
 27. Smith, R.J., Heywood, M.I.: Evolving Dota 2 shadow fiend bots using genetic programming with external memory. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'19, pp. 179–187. ACM, New York, NY, USA (2019)
 28. Smith, R.J., Heywood, M.I.: Interpreting tangled program graphs under partially observable Dota 2 invoker tasks. *IEEE Trans. Artif. Intell.* **5**(4), 1511–1524 (2024)

29. Stanley, K.O., Miikkulainen, R.: Competitive coevolution through evolutionary complexification. *J. Artif. Int. Res.* **21**(1), 63–100 (2004)
30. Sutton, R.S., Barto, A.G.: *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA (2018)
31. Teller, A.: The evolution of mental models. In: *Advances in Genetic Programming*, vol. 1. The MIT Press (1994)
32. Trappenberg, T.P.: *Fundamentals of Computational Neuroscience*. Oxford University Press UK (2002)
33. Turner, A.J., Miller, J.F.: Recurrent cartesian genetic programming of artificial neural networks. *Genet. Program. Evol. Mach.* **18**(2), 185–212 (2017)
34. Vithayathil Varghese, N., Mahmoud, Q.H.: A survey of multi-task deep reinforcement learning. *Electronics* **9**(9) (2020)
35. Wineberg, M., Oppacher, F.: The benefits of computing with introns. In: *Proceedings of the 1st Annual Conference on Genetic Programming*, pp. 410–415. MIT Press, Cambridge, MA, USA (1996)

Chapter 6

Automatic Design of Autoencoders Using NeuroEvolution



Alexy de Almeida, Nuno Lourenço, and Penousal Machado

Abstract AutoEncoders (AEs) are widely used for efficient data compression and feature extraction. However, finding the most appropriate architecture and set of parameters that result in the most effective approach for a certain domain is not an easy task. This study is inspired by the significant potential of using NeuroEvolution (NE) to optimize and develop AEs through evolutionary algorithms. For that purpose, a framework based on Fast Deep Evolutionary Network Structured Representation (Fast-DENSER) is introduced for automatically training and evolving an AutoEncoder (AE), aimed at offering Deep Neural Network (DNN) architectures potentially adept at addressing a range of tasks such as data denoising, anomaly detection, and high-resolution image reconstruction. The methodology involves intricate optimization and time-intensive exploration for identifying an optimal architecture tailored to a specific task, with the susceptibility to overfitting influencing result interpretation. Our approach involves training and evolving the encoder and decoder components of the AE separately. The experimental findings showcase diverse architectures, shedding light on the types of AE architectures favorable and unfavorable to effective training and evolution. The experiments also demonstrate the use of different fitness functions and the metrics that are most and least suited to image reconstruction problems, along with their influence on the training and evolution process.

6.1 Introduction

Nowadays, Artificial Intelligence (AI) and big data are becoming commodities, and with the volume and complexity of data generated becoming increasingly difficult to deal with, there is a demand for tools and procedures to deal with data representation

A. de Almeida · N. Lourenço (✉) · P. Machado
CISUC/LASI, DEI, University of Coimbra, Coimbra, Portugal
e-mail: naml@dei.uc.pt

A. de Almeida
e-mail: adenis@student.dei.uc.pt

P. Machado
e-mail: machado@dei.uc.pt

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2025
S. M. Winkler et al. (eds.), *Genetic Programming Theory and Practice XXI*, Genetic and Evolutionary Computation, https://doi.org/10.1007/978-981-96-0077-9_6

and processing. In the last few years, the emergence of AEs has been able to address some of these issues thanks to their ability to extract learning patterns from raw data. They have been used in a variety of scenarios, ranging from denoising a medical image to reconstructing a low-sized image to a super-resolution one or even detecting anomalies in security camera footage, which we explored more in depth in Sect. 6.2.

However, building AEs is difficult. Finding a model that delivers good results can be challenging. This difficulty arises not only from the complexity of designing an effective architecture but also from the high computational costs associated with training large neural networks. Moreover, AEs are susceptible to overfitting, and the outcomes obtained do not always accurately pinpoint potential issues in the entire process. Srivastava et al. [11] and Liang and Liu [9] address this problem by using Dropout layers, one of many regularizer techniques, which drop a proportion of the neurons in a layer during the training step.

The main goal of this work is to analyze the possibility of using an Evolutionary Machine Learning (EML) model for the automatic generation of AEs. By extending the use of the Fast-DENSER framework proposed by Assunção et al. [2–4], we want to find suitable models for diverse tasks of image reconstruction. We begin by designing and fine-tuning multiple AEs by fixing the decoder part, evolving only the encoder. Then, we replicate the same idea by fixing the encoder architecture and evolving the decoder. Lastly, taking into account the previous experiments, we evolve a convolutional AE. The motivations and ideas behind these experiments are discussed and explained in Sect. 6.4.

In Sect. 6.2, we perform a short survey of denoising, image reconstruction, anomaly detection techniques, and methods to guide the evolutionary process in DNNs. In Sect. 6.3, we present the Fast-DENSER framework and explain how it operates. In Sect. 6.4, we detail the adaptations made to allow the evolution of an AE. We also present the experiments on several datasets using multiple AEs with different architectures and experimental designs, discussing and analyzing the presented experimental results. This section is divided into three parts: Fixed-Decoder architectures in Subsect. 6.4.1, Fixed-Encoder architectures in Subsect. 6.4.2, and Convolutional AE architectures in Subsect. 6.4.3. Lastly, some conclusions and future research are indicated in Sect. 6.5.

6.2 State of the Art

In this section, we identify already existing solutions related to AEs and their diverse applications.

One way to reconstruct images is using a method called Single Image Super-Resolution (SISR), which reconstructs a super-resolution image from a single low-resolution image. Dong et al. [5] propose a model called Super-Resolution Convolutional Neural Network (CNN), which is a fully feed-forward, fully CNN. The only pre-processing they do is upscaling a single low-resolution image using bi-cubic interpolation, then proceeding with three operations that will represent the

CNN: (i) extracting patches from the upscaled image and representing each patch as a high-dimensional vector; (ii) mapping each high-dimensional vector onto another high-dimensional vector, representing a high-resolution patch; and (iii) generating the final high-resolution image by aggregating the two previous operations. The results are very promising because they achieve superior performance compared to state-of-the-art methods.

Ahn and Yim [1] also use **CNNs**, but they apply skip connections with layer groups, resulting in processed data from the hidden layers being transmitted to the output more efficiently. They group the layers and connect their output to input data with a skip connection, which they find advantageous for two things: (i) preserving the features of the input image to be learned for reconstructing a super-resolution version of that image; and (ii) alleviating the problem of data extinction existing with ReLU operations.

In the field of image denoising, Gondara [6] applies a combination of **CNN** and **Denoising AutoEncoder (DAE)**, using two datasets: (i) a mini-MIAS database of 322 1024×1024 mammograms; and (ii) a dental radiography database of 400 1935×2400 cephalometric X-ray images, all of them being resized to 64×64 . Then, they added Poisson and Gaussian noise to a flattened dataset instead of adding manually to each single image. They show very promising results using a **Convolutional Denoising AutoEncoder (CNN DAE)** versus other conventional denoising methods. As expected, the more noise is introduced, the harder it is to reconstruct perceptible images, but even by adding so much noise that it makes the image unrecognizable, the model is able to reconstruct partial generation of real images close to the original ones.

Kulkarni et al. [8] also do something very similar by resizing their FFHQ-Face dataset of 70 000 images to 128×128 and adding Poisson noise. Their results show a growth of **Peak Signal-to-Noise Ratio (PSNR)** for every epoch, varying between 35 and 36 until 40 epochs and slowly starting to go up after, compared to other already pre-trained models that show a value of **PSNR** going from 26 to 33. They also use **Mean Squared Error (MSE)** to measure the error/loss between iterations, which shows a slight decrease as the number of epochs increases.

Anomaly detection works follow a similar goal to denoising and image reconstruction. One example is anomaly detection in surveillance videos, which Mishra and Jabin [10] explain is possible by comparing the reconstructed frames with the original frames and identifying the main differences. Since they are dealing with video data, they use an architecture made of two 3D convolution layers, three 2D **Long Short-Term Memory (LSTM)** convolution layers, and two 3D deconvolution layers, with a mix of tanh, ReLU, and sigmoid activation functions. Their CUHK Avenue dataset contains 16 training and 21 testing video clips, which have 15,328 training videos and 15,324 frames in total. They preprocess the dataset by extracting frames from the video clips at 15 fps, resizing them to 227×227 , normalizing the pixel values from 0 to 1, and then creating sequences of 10 frames each. The architecture previously detailed obtained the best results, achieving an **Area Under the Curve (AUC)** of 86.4 and 88.9%. Even though they are good scores, they are still

below the results of other non-AE techniques mentioned in their research, which go above an AUC 0.93.

Suganuma et al. [12] exploit the potential of standard Convolutional Autoencoder (CAE)s for image restoration by applying an evolutionary algorithm to search for optimal CAE architectures. They conducted two different types of experiments, one using denoising and the other using image inpainting [14]. For the denoising experiments, using three noise levels and the BD200 dataset, they were able to achieve PSNR values ranging from 24.83 to 28.23 and Structural Similarity (SSIM) values ranging from 0.6636 to 0.8047. For the inpainting experiments, using three masking patterns and three datasets, the PSNR values ranged from 16.2 to 40.4 and the SSIM values ranged from 0.610 to 0.969.

Hashim et al. [7] propose an evolutionary AE to develop an architecture designed to handle unlabeled sensor data. Their best model shows an accuracy of 0.9496, f1 of 0.9982, precision of 0.9506, recall of 0.9814, and loss of 0.0398, performing better than the compared models in their work.

6.3 Fast-DENSER

Combining the concepts of Genetic Algorithm (GA) and Dynamic Structured Grammatical Evolution (DSGE), Deep Evolutionary Network Structured Representation (DENSER) [2] is a novel evolutionary approach for the automatic generation of DNNs. An extension of the original framework called Fast-DENSER was then proposed [4], with the main advantage of offering a speedup of 20x without compromising the performance of the found solutions [4]. The algorithm of this framework aims at searching for DNN topologies and hyper-parameters. Figure 6.1 depicts a condensed view of the framework, with the following numbered steps:

1. A configuration and grammar files are loaded. All the evolutionary and network structure parameters can be defined in the configuration file, while the network parameters can be configured in a grammar file in human-readable text format, allowing any user to easily modify the setup of the experiences.
2. The first generation creates a population of DNNs based on the parameters defined in the configuration files.
3. The population is evaluated so that the fittest network of that generation is obtained.
4. The fittest neural network is selected, then saved along with all the other fittest neural networks from each generation. The best fitness from all generations is also saved, and if the best fitness of the current generation is better than the overall best fitness, then it is updated. The phenotype of the population and the files of the best individual are also kept.
5. The population from the previous generation is loaded, and the fittest individual is used to generate offspring. Afterward, the new population is made up of that

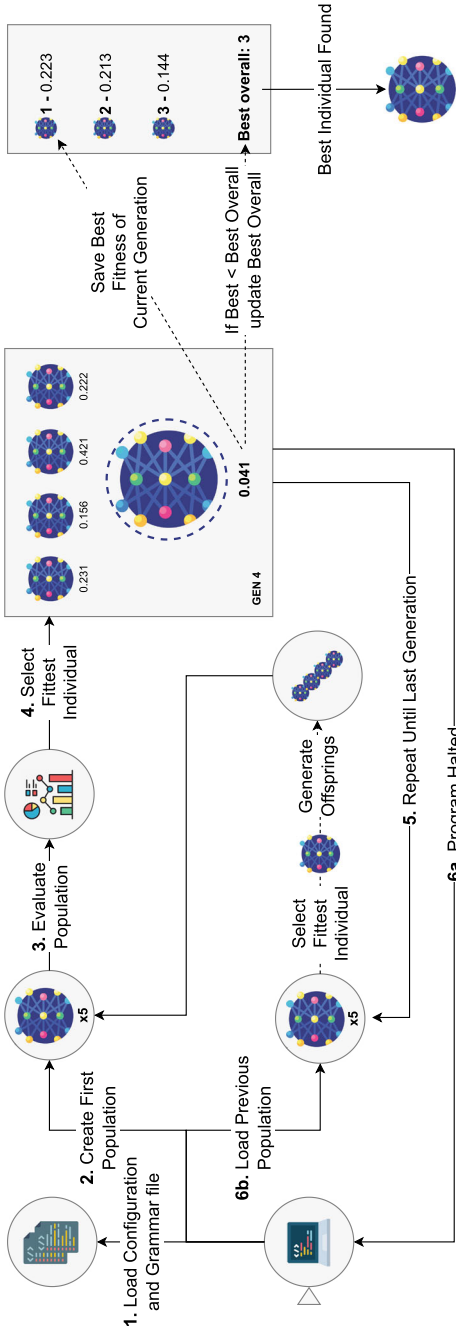


Fig. 6.1 Fast-DENSER architecture

fittest individual and its offspring. The program then repeats steps 5, 3, and 4 until the last generation.

6. If the program is halted, the algorithm goes back to the start (6a). The files need to be loaded again, and this time, the program loads the last saved generation (6b).

The configuration files allow the grammar to have two representations:

- (i) At the **GA**-level represents the sequence of evolutionary units that form the network. Those units are part of the encoder, bottleneck, and decoder of an **AE**, and they point to a grammar start symbol at the **DSGE**-level. At this level are also defined the minimum and maximum number of layers for each structure of the **AE**. An example of a **GA** structure can be: {"networkstructure": [{"encoder", 1, 20], ["bottleneck", 1, 1], ["decoder", 1, 20]], "output": "fully connected", "networkstructinit": ("encoder": [2, 3, 4], "bottleneck": [1], "decoder": [2, 3, 4])}. With this structure, we are specifying that we want an **AE** to have between 1 and 20 layers in the encoder, 1 layer in the bottleneck, and 1–20 layers in the decoder. The last layer of the network must be a Fully Connected and the initial population of **AEs** must start with 2–4 layers, both in the encoder and decoder, and 1 layer in the bottleneck.
- (ii) In the **DSGE**-level, the parameters are codified in a **Backus-Naur Form (BNF)** grammar. These parameters represent the layers, learning, variable types, and range of values, among other aspects of the evolution's macrostructure. In subsequent sections, a grammatical representation of our experiments is provided in their experimental setup.

Following the work done by Assunção et al. [2–4], we adapted the framework to be able to automatically design and evolve multiple **AEs** with any kind of architecture. Currently, the main purpose is to do image reconstruction and, later on, image denoising. The adaptations made also allow to isolate the encoder or decoder from the evolution process to obtain an **AE** with a fixed part and an evolved part. To ensure the validity of the evolved architectures, several mechanisms were progressively implemented through the different architecture types, which will be discussed in the next section and subsections.

6.4 Evolving AutoEncoders

In order to evolve an **AE**, we make use of the grammar and configuration files present in the original framework, as explained in Sect. 6.3. The user can manually configure the parameters to ensure that the output image has the same shape as the input image, while the algorithm adjusts the evolved architectures to prevent having an incorrect disposition of layers, such as eliminating Max Pooling layers when no more downsampling operations can be made.

Algorithm 6.1 Fast-DENSER Evolutionary Algorithm**Require:** $pop_size \geq 0, generations > 0, 0 \leq crossover \leq 1, 0 \leq mutation \leq 1$

```

1:  $gen \leftarrow 0$ 
2:  $pop \leftarrow \{\}$ 
3:  $best\_fit \leftarrow \text{NULL}$ 
4: while  $gen < n\_generations$  do
5:    $best\_fit\_gen \leftarrow 0$ 
6:   if  $gen == 0$  then
7:     for  $i$  in  $pop\_size$  do
8:        $pop \leftarrow pop \cup individual$ 
9:        $i \leftarrow i + 1$ 
10:    end for
11:  else if  $gen > 0$  then
12:     $pop \leftarrow pop \cup parent$ 
13:    for  $i$  in  $pop\_size$  do
14:       $pop \leftarrow pop \cup mutation(parent)$ 
15:       $i + 1$ 
16:    end for
17:  end if
18:  for  $individual$  in  $pop$  do
19:     $evaluate(individual)$ 
20:    if  $fit(individual) > best\_fit\_gen$  then
21:       $best\_fit\_gen = fit(individual)$ 
22:       $parent \leftarrow individual$ 
23:    else if  $best\_fit = \text{NULL}$  or  $fit(individual) > best\_fit$  then
24:       $best\_fit = fit(individual)$ 
25:    end if
26:  end for
27: end while

```

We began the first set of experiments by evolving AEs with Fully Connected architectures for the MNIST and Fashion-MNIST datasets using different parameters like the kernel-initializer, kernel-regularizer, bias, number of units, loss functions, output functions, data augmentation, number of generations, and train time. While the results were good for the MNIST dataset, they were not as good for the Fashion-MNIST since the models struggled to reconstruct the images, displaying blurry outputs. We concluded that a model using a Fully Connected architecture was not enough for reconstructing the images properly and was not efficient memory- and time-wise.

In the next step, we designed and tried to evolve architectures with a higher variety of layers, such as Convolution, Max Pooling, and Upsampling. Since the original framework was not intended for evolving AEs, the main problem found was that the configurations and the algorithm only worked for complete Fully Connected or Fully Connected decoder topologies, making it impossible to evolve complete Convolutional ones. This is because a Flatten layer would always be placed before the first Fully Connected layer, converting the shape of the last 3D layer to 1D. To maintain continuity between the sequence of layers in the architecture, the shape sizes would need to match, but there were two flaws in the original mechanism:

(i) it would not do a conversion from 1D to 3D layer shapes; (ii) it would not match the output image size to the input image if other layers than Fully Connected were used in the decoder.

To address this issue, we began by implementing the first version of our current mechanism. We created a bottleneck comprised of 3 layers: (i) a Flatten layer to flatten the number of neurons from the previous layers from 3D to 1D, (ii) a Fully Connected layer with a manually defined number of neurons (mandatorily a potency of two); and (iii) a Reshape layer to match the 3D shape of the decoder layers. This solved the first problem of not being able to convert from 1D to 3D shapes, but it started to introduce fixed components to an evolutionary structure, while still not solving the second problem.

This problem gave us inspiration for evolving architectures with fixed components, such as the encoder or the decoder. By doing so, we could also look for solutions to the remaining problem and analyze how to make the evolution completely free of fixed components.

6.4.1 Fixed-Decoder Image Reconstruction

The challenge posed by a fixed structure in the decoder served as the primary motivation to evolve an **AE** with a predetermined topology. The aim was to evolve an **AE** wherein the decoder segment remained fixed, thereby examining its influence on the encoder architecture. Additionally, the objective was to discern the adjustments necessary to transition from a fixed structure to an autonomously designed one. Table 6.1 outlines the parameters employed for the 40 experimental setups devised to address this issue. The experiments were executed according to the following procedure:

- (i) Experiments 1–7 involved a trial-and-error approach to using a different number of neurons for the Fully Connected layer in the previously mentioned bottleneck. Additionally, different loss functions (e.g., **MSE**, **Kullback–Leibler divergence**

Table 6.1 Fixed-decoder experimental setup. Run: Number of runs, Pop: Population Size, M: MNIST, F: Fashion-MNIST, C: CIFAR-10, CF: CIFAR-100 (Fine), S: SVHN, T: tiny-ImageNet, Loss: Loss function, Fitness: Fitness function, Dec: Decoder

Run	Gens	Pop	Early stop	Datasets	Loss	Fitness	Dec
1–7	400	5	No	M-F-C	Multiple	Multiple	1
8–10	400	5	No	C	MSE	Multiple	1
10–12	400	5	Yes	M-F-C	MSE	Multiple	1
13–22	400	5	Yes	F-C	MSE	Custom	1
23–28	400	5	Yes	C	MSE	Custom	1
29–34	10	2	Yes	C	MSE	Custom	9
35–40	100	5	Yes	M-F-C- CF-S-T	MSE	Custom	9

(KL divergence), or SSIM) and fitness functions (e.g., MSE, PSNR, SSIM, or a combination of these) were explored. Mixed topologies in the encoder—utilizing Fully Connected, Convolutional, or a hybrid of both—were tested, along with adjustments to filter values in the decoder layers. Other variations included manually setting the first encoder layer to be Convolutional and experimenting with different hyper-parameters for the layers. However, these attempts did not yield satisfactory results;

- (ii) Experiments 8–10 employed a convolutional topology in the encoder and increased the size of the fully connected layer in the bottleneck. Additionally, some experiments, not depicted in Table 6.1, were parallelized. These parallel experiments involved varying percentages for dataset training and testing, as well as adjusting learning rates for different loss functions, with Adam demonstrating superior performance, followed by RMSProp. Attempts were also made to enhance filter values in the decoder layers; however, the results remained unsatisfactory;
- (iii) Experiments 10–12 retained the previous convolutional topology in the encoder and introduced an early stop mechanism that would halt the training of the current network if the loss value failed to improve after two epochs. We started to observe improvements to the output images, as they were starting to become less blurry and more accurate;
- (iv) Experiments 13–22 involved altering the Convolution layer filter sizes in the encoder, the number of neurons in the bottleneck and modifying the fitness function, termed “Custom”. This function included the MSE, PSNR, SSIM, and accuracy values with different percentages. While one of these experiments yielded promising results, the subsequent replication of the same model design in five additional experiments proved disappointing;
- (v) Experiments 23–28 used the opposite approach to previous experiments by varying the Convolution filter values in the decoder while keeping the range of values consistent in the encoder and bottleneck. Despite achieving some accuracy, the results kept displaying blurriness;
- (vi) Experiments 29–34 solved the blurriness problem by introducing a new Fixed-Decoder mechanism while also eliminating the need for a bottleneck. Based on previous experiments, we determined that Batch Normalization layers were efficient in speeding up the training process while enhancing the AE model quality. Upon insertion of the Convolution Transpose layer, the algorithm automatically computes the kernel and stride values for that layer. Additionally, based on the resulting layer’s shape, the algorithm determines the required number of UpSampling layers to align the output and input shapes. The quantity of UpSampling layers could be adjusted using a window size value utilized in the calculation of kernel and stride parameters.

Experiments 35–40 follow the same logic as previous experiments but were run for a higher number of generations, populations, and datasets, yielding the best results. Figure 6.3 shows the grammar used for evolving the encoder. x and y values represent the number of filters for the convolution layers, with different values being used for different experiments (Fig. 6.2).

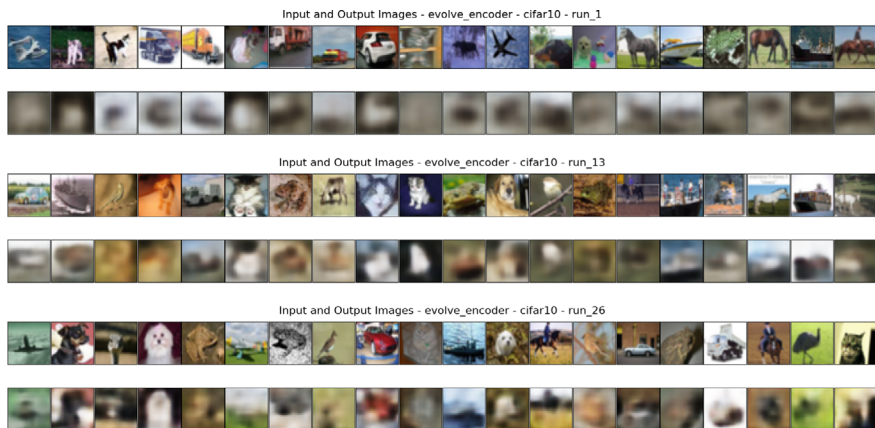


Fig. 6.2 Experiments 1, 13, and 26—Input and output images displaying the blurriness problem, CIFAR-10 dataset. The reconstructed images get progressively better

```

< encoder > ::= < convolution >
               | < batch - norm >
               | < dropout >
               | < pooling >
< convolution > ::= layer : conv[num - filters,int,1,x,y]
                  [filter - shape,int,1,2,5][stride,int,1,1,3]
                  < padding > < activation - function > < bias >
< padding > ::= padding : same
               | < padding : valid >
< bias > ::= bias : True
           | bias : False
< activation - function > ::= act : relu

```

Fig. 6.3 CFG of the fixed-decoder architectures—Experiments 35–40

As previously mentioned, the fitness function was adjusted with the goal to capture different aspects of the images for a more accurate output reconstruction. According to Wang et al. [13], the most simple and used metrics for image quality assessment are the **MSE**, the **SSIM**, and the **PSNR**, being appealing for their simplicity and convenience.

(i) The **MSE** measures the average squared difference between the original image and the reconstructed image, but because it treats all pixels the same way, it does not correlate well with image quality in the human perception. This metric is the most widely used in our work because not only it is used as the loss function, but it also aids in computational and time efficiency; (ii) the **SSIM** compares the structural similarity, such as the luminance, contrast, and structure of the images, so it is more aligned with how humans perceive image quality; (iii) the **PSNR** measures the ratio between the maximum possible power of the input and the power of the difference

Table 6.2 Experimental parameters of the fixed-decoder architectures—Experiments 35–40

Evolutionary parameters	Value
Number of generations	100
Population size	5
Max epochs	10000
Mutation parameters	Value
Add layer	45%
Reuse layer	45%
Remove layer	45%
DGSE layer	20%
Macro layer	20%
Train longer	20%
Train parameters	Value
Train time	10 min
Loss function	Mean squared error
Fitness metric	Custom
Network structure	Value
Initial number of encoding layers	2–4
Total number of encoding layers	2–10
Decoding layers	Fixed

between the output and input, but suffers from the same problem as the [MSE](#). Its values range from 0 to infinity, but in this work we established a maximum value of 100 when the [MSE](#) is 0, which means that the reconstructed image is exactly the same as the input image.

These metrics would not work well if used alone due to the wide nature of images, so we combined them together to reduce the error in image reconstruction. We named it “custom fitness” and it is calculated as follows:

$$\text{fitness} = \text{MSE}(34\%) + \text{SSIM}(33\%) + \text{PSNR}(33\%)$$

Since we aim to achieve the lowest fitness value, we created the fitness in a way that a lower [MSE](#) and higher [PSNR](#) and [SSIM](#) values contribute positively to its value.

Table 6.2 displays the evolutionary and network parameters utilized in these six experiments. While the majority of previous experiments (1–34) were conducted over 400 generations, our primary focus on addressing the blurriness issue prompted a reduction to 100 generations. This adjustment was influenced by the observation that experiments 29–34 demonstrated promising outcomes within just 10 generations, with a population size of 2, making the process less time-consuming.

Table 6.3 Fixed-decoder results for experiments 35–40. x and y units refer to the number of filters for the convolution layers in the encoder. **Time showed is the average for every generation, for the whole population of individuals.** Enc—Encoder

Enc	Units (x-y)	Fitness (↓)	FID (↓)	PSNR (↑)	SSIM (↑)	MSE (↓)	Time (↓)
35	16–64	0.087	562.483	73.908	0.878	0.004	09 min 38 s
36	16–128	0.088	562.296	73.557	0.929	0.003	09 min 13 s
37	16–256	0.113	564.777	70.870	0.792	0.050	10 min 01 s
38	16–64	0.095	562.244	72.025	0.905	0.007	07 min 44 s
39	16–64	0.118	562.601	66.587	0.730	0.022	07 min 25 s
40	16–64	0.098	562.356	70.961	0.855	0.007	08 min 13 s

Table 6.4 Average results from Table 6.3 containing fixed-decoder results for experiments 35–40. MNIST, Fashion-MNIST, CIFAR-10, CIFAR-100 (Fine), SVHN, and tiny-ImageNet datasets. **Time showed is the average for every generation, for the whole population of individuals**

Units (x-y)	Fitness	FID	PSNR	SSIM	MSE	Time
Variable	0.1	562.793	71.318	0.848	0.016	08 min 42 s

6.4.1.1 Experimental Results

Table 6.3 presents the metric outcomes of the top-performing AEs discovered in experiments 35–40, wherein each experiment utilized a distinct encoder while employing the same decoder. The fitness, Frechet Inception Distance (FID) score, PSNR, SSIM, MSE, and individual training times were computed across six datasets as listed in Table 6.1. In essence, for each experiment, the reported values represent the average results derived from these six datasets.

The computation of the FID score occurred only upon the completion of the final phase of this work, which involved evolving a complete AE. The motivation for using it is explained in Sect. 6.4.1. Given the close proximity of the fitness and FID score values across each experiment, it becomes more interesting to examine the PSNR, SSIM, and MSE values, alongside the average training time for each individual. While AE 35 exhibited the best PSNR and AE 36 exhibited the best SSIM and MSE, they required more individual training time compared to AE 38, which has closely aligned values. As a result, AEs 36 and 38 emerge as a particularly noteworthy candidates for analysis, warranting an examination of its network structure.

Looking at Fig. 6.4, we can see a close quality in terms of image reconstruction, except for the third experiment, whose quality is not accurately reflected in the metrics.

Since our experiments were conducted using Keras and TensorFlow, we made use of the Python package Visualkeras to visually display the resulting architectures.

Figure 6.5 depicts the Fixed-Decoder architecture common to every AE from experiments 35–40. Figure 6.6 shows the encoder architecture of experiment 36 on

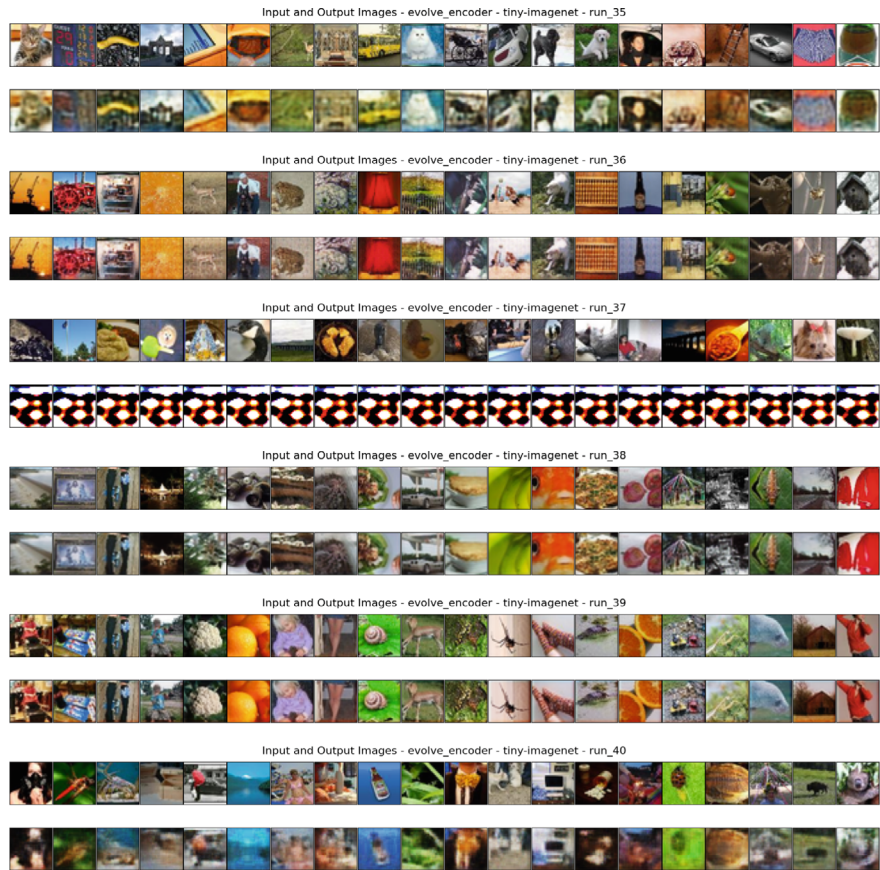


Fig. 6.4 Fixed-encoder—Experiments 35–40—Input and output images, tiny-ImageNet dataset

the left and experiment 38 on the right. It is interesting to observe the noticeable number of Batch Normalization layers on experiment 38 compared to experiment 36, which displays none. These layers may play an important role in speeding the training process, which aligns with the results shown in Table 6.3. Moreover, an excessive number of Convolution layers may potentially lead to inferior outcomes, which was observed in other architectures not displayed here. This gives us a first indication of the type and quantity of layers to build a better model. Other experiments also showed us architectures where Batch Normalization layers were predominant, with a Batch Normalization, Convolution and Dropout ratio of 2:2:1 or 3:2:1.

Fig. 6.5 Fixed-decoder decoder architecture, Experiments 35–40. *Note* Each color represents a layer; Green: Input—Yellow—Batch normalization; Red—Convolution; Pink—Dropout; Blue—Convolution transposed; Magenta—UpSampling

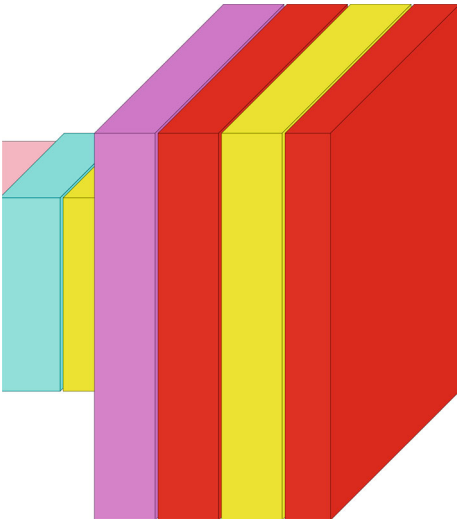
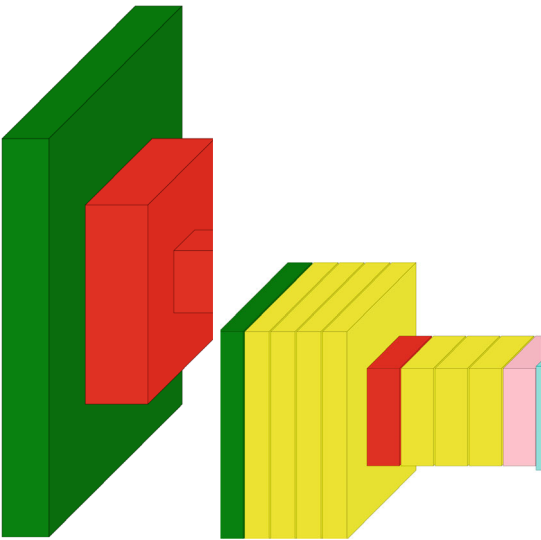


Fig. 6.6 Fixed-decoder encoder architectures, tiny-ImageNet dataset. *Note 1* The scale is different in each image. *Note 2* Each color represents a layer; Green: Input—Yellow—Batch normalization; Red—Convolution; Pink—Dropout; Blue—Convolution transposed; Magenta—UpSampling



(a) Experiment 36 (b) Experiment 38

6.4.2 Fixed-Encoder Image Reconstruction

Having successfully evolved encoders with fixed decoders, our aim shifted toward exploring the inverse scenario: evolve decoders and keep the encoder fixed. To this end, 7 experimental designs were established. To evolve a decoder, we took the Fixed-Decoder mechanism and adjusted it so that it calculates the kernel and stride values


```

< decoder >          ::= < convolution > < batch - norm > < convolution >
                        < dropout > < batch - norm
< encoder1 >         ::= < convolution1 > < batch - norm > < convolution1 >
                        < dropout > < batch - norm
< encoder2 >         ::= < convolution2 > < batch - norm > < convolution2 >
                        < batch - norm
< convolution >       ::= layer : conv[num - filters,int,1,x,y]
                        [filter - shape,int,1,2,5][stride,int,1,1,3]
                        < padding > < activation - function > < bias >
< convolution1 >      ::= layer : conv[num - filters,int,1,64,64]
                        [filter - shape,int,1,2,5][stride,int,1,1,3]
                        < padding > < activation - function > < bias >
< convolution2 >      ::= layer : conv[num - filters,int,1,32,32]
                        [filter - shape,int,1,2,5][stride,int,1,1,3]
                        < padding > < activation - function > < bias >
< padding >          ::= padding : same
                        | < padding : valid >
< bias >              ::= bias : True
                        | bias : False
< activation - function > ::= act : relu

```

Fig. 6.7 CFG for AE Fixed-Decoder architectures

based on the last layer to create a Convolution Transpose layer, a variable number of UpSampling layers and a final Convolution layer to match the input image shape. These layers do not count toward the total number of layers in the decoder but serve solely to resize the output to the desired dimensions.

Drawing from prior experiments, we established two Fixed-Encoder architectures, as depicted in Fig. 6.7. The sole distinction between them lies in the inclusion or omission of a Dropout layer, allowing us to evaluate the occurrence of overfitting and its consequential impact on image reconstruction.

Table 6.5 shows the evolutionary and network parameters for two different types of experiments, where the only difference is the number of runs (100 for the first vs 400 for the second). The previous experiments were run for 100 generations, so we maintained that consistency for the first experiment design, but we wanted to observe how more runs could also impact the algorithm in finding better AEs.

6.4.2.1 Experimental Results

Table 6.7 shows marginally worse results for the fitness and MSE values compared to the Fixed-Decoder results shown in Table 6.4. Every population takes slightly more time to train, while the PSNR and SSIM values are significantly worse. The lack of variance in the FID score shown in Table 6.3 suggests that it may not be an effective metric for assessing the quality of image reconstructions.

Observing Fig. 6.8, it is evident that the issue of blurriness persisted across all experiments, except for experiment 6, which yielded superior results. A closer exam-

Table 6.5 Experimental parameters of the fixed-encoder architecture

Evolutionary parameters	Value
Number of generations	100 and 400
Population size	5
Max epochs	10000
Mutation parameters	Value
Add layer	45%
Reuse layer	45%
Remove layer	45%
DGSE layer	20%
Macro layer	20%
Train longer	20%
Train parameters	Value
Train time	10 min
Loss function	Mean squared error
Fitness metric	Custom
Network structure	Value
Encoding layers	Fixed
Initial number of decoding layers	2–4
Total number of decoding layers	2–10

Table 6.6 Fixed-encoder results for experiments 1–7. x and y units refer to the number of filters for the convolution layers in the encoder. **Time showed is the average for every generation, for the whole population of individuals.** Enc—Encoder, Dec—Decoder

Gen	Enc	Dec	Units (x-y)	Fitness (↓)	FID (↓)	PSNR (↑)	SSIM (↑)	MSE (↓)	Time (↓)
100	1	1	32–64	0.117	562.686	66.236	0.597	0.016	08 min 38s
100	1	2	32–128	0.122	562.214	65.317	0.632	0.021	10 min 13 s
100	1	3	64–256	0.114	562.267	67.109	0.651	0.014	09 min 50 s
100	1	4	64–256	0.115	562.298	66.728	0.681	0.015	08 min 43 s
400	2	5	32–64	0.123	561.471	65.585	0.53	0.028	08 min 48 s
400	2	6	32–128	0.104	561.972	69.545	0.819	0.009	09 min 00 s
400	2	7	64–256	0.111	562.064	67.643	0.714	0.013	09 min 00 s

Table 6.7 Average results from Table 6.6 containing fixed-encoder results for experiments 1–7. MNIST, Fashion-MNIST, CIFAR-10, CIFAR-100 (Fine), SVHN, and tiny-ImageNet datasets. **Time showed is the average for every generation, for the whole population of individuals**

Units (x-y)	Fitness	FID	PSNR	SSIM	MSE	Time
Variable	0.115	562.139	66.880	0.661	0.017	09 min 10 s

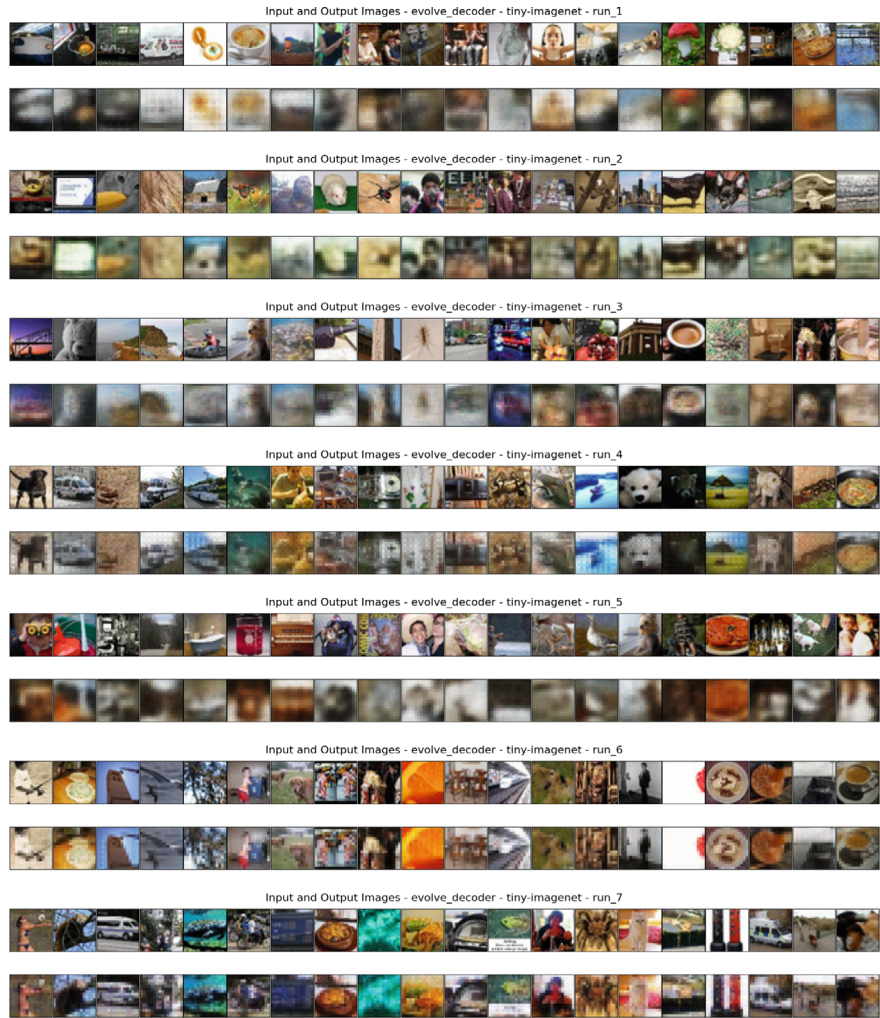


Fig. 6.8 Fixed-encoder—Experiments 1–7—Input and output images, tiny-ImageNet dataset

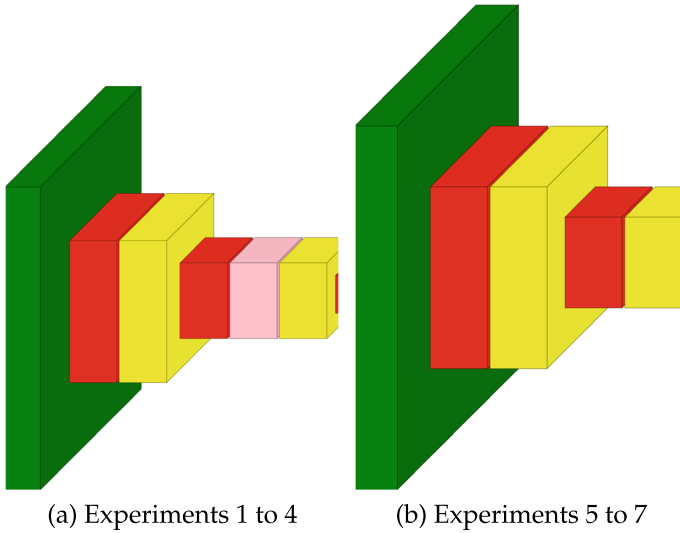


Fig. 6.9 Fixed-encoder encoder architectures. *Note 1* The scale is different in each image. *Note 2* Each color represents a layer; Green: Input—Yellow—Batch normalization; Red—Convolution; Pink—Dropout; Blue—Convolution transposed; Magenta—UpSampling

ination of Table 6.6 reveals a strong correlation between the metric scores and the quality of reconstructed images, highlighting PSNR, SSIM, and MSE as reliable metrics for this context. However, despite designing the encoders based on insights from prior experiments in the Fixed-Decoder Section, it is important to develop alternative encoder designs to explore potentially improved outcomes.

Figure 6.9 shows the encoder architecture of the first type of experiment on the left and of the second type on the right.

Figure 6.10 shows the decoder architecture of experiment 2 on the left, which belongs to the first type of experiment; and, on the right, it shows the decoder architecture of experiment 6, belonging to the second type of experiment. Similar to the findings in Fixed-Decoder experiments, an excessive number of Convolution layers seems to result in poorer outcomes. In other Fixed-Encoder architectures not displayed here, there is also a notable presence of Dropout layers, which noticeably affect the quality of the reconstructed image. Despite an equal likelihood of including Convolution, Batch Normalization, and Dropout layers, evolution tends to favor architectures characterized by a higher prevalence of Batch Normalization layers.

6.4.3 Convolutional AutoEncoder Image Reconstruction

Upon successfully isolating and evolving both the encoder and decoder components, we started to fully evolve an AE with a convolutional topology. Although still in the

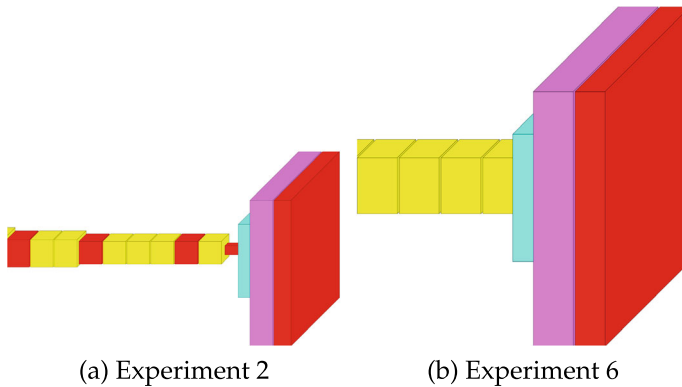


Fig. 6.10 Fixed-encoder decoder architectures, tiny-ImageNet dataset. *Note 1* The scale is different in each image. *Note 2* Each color represents a layer; Green: Input—Yellow—Batch normalization; Red—Convolution; Pink—Dropout; Blue—Convolution transposed; Magenta—UpSampling

early stages, the outcomes of previous experiments have been promising. The architectures from previous experiments were helpful in making us understand what kind of AE architectures reconstruct images the best. Notably, architectures incorporating a mix of Convolution, Batch Normalization, UpSampling, Dropout, and Convolution Transpose layers, with a higher proportion of Batch Normalization layers relative to Convolution layers, along with minimal Dropout layers, have demonstrated a strong correlation with improved reconstruction quality. This kind of architecture is showcased through the grammar in Fig. 6.11. Additionally, those results were instrumental in configuring several parameters, particularly the number of filters for the convolution layers and the number of generations. Increasing the number of filters tended to degrade performance, whereas a higher number of generations led to slightly improved results. However, these improvements were often negligible, making the trade-off potentially unwarranted.

Additionally, the integration of the FID score into our evaluation aims to ascertain its potential as a better fitness function compared to the previously used ones, particularly in cases where the fitness was not useful for image quality assessments. Since it was only implemented at this stage, we could not yet draw any conclusions from the values from previous experiments. We also maintained the same decoder mechanism from previous experiments to assure that the output layer shape matches the input one.

In this experiment setup, we increased the maximum number of epochs from 10,000 to 100,000 to accommodate the extended training time required for each individual due to the inclusion of the FID score as a quality assessment metric. All other parameters remained unchanged from previous experiments. However, unlike previous experiments, we evolved and trained 5 models with the same experimental configuration, shown in Table 6.8 for the CIFAR-100 (Fine) dataset only. This dataset was chosen for its simplicity in training multiple models simultaneously

```

< encoder > ::= < convolution >
              | < batch – norm >
              | < dropout >
< decoder > ::= < convolution >
              | < batch – norm >
              | < dropout >
< convolution > ::= layer : conv[num – filters,int,1,16,64]
                  [filter – shape,int,1,2,5][stride,int,1,1,3]
                  < padding > < activation – function > < bias >
< padding > ::= padding : same
              | < padding : valid >
< bias > ::= bias : True
          | bias : False
< activation – function > ::= act : relu

```

Fig. 6.11 CFG for Convolutional AE architectures

Table 6.8 Experimental parameters of Convolutional AE architectures

Evolutionary parameters	Value
Number of generations	400
Population size	5
Max epochs	100000
Mutation parameters	Value
Add layer	45%
Reuse layer	45%
Remove layer	45%
DGSE layer	20%
Macro layer	20%
Train longer	20%
Train parameters	Value
Train time	10 min
Loss function	Mean squared error
Fitness metric	FID and custom
Network structure	Value
Initial number of encoding layers	2–4
Initial number of decoding layers	2–4
Total number of encoding layers	2–10
Total number of decoding layers	2–10

while being more complex than the MNIST dataset, as the incorporation of the **FID** score alongside more complex datasets significantly affects CPU usage.

6.4.3.1 Experimental Results

As noted in Table 6.9, experiment 2 stands out as the only one maintaining comparable results to those of previous experiments. However, overall, the metrics indicate a decline in performance across the board.

Table 6.10 presents the average values extracted from Table 6.9. In contrast to outcomes from previous experiments, the results are notably worse. This reinforces the inadequacy of the **FID** score as both a fitness quality assessment metric and its detrimental impact on training time and CPU usage, rendering it unsuitable for image reconstruction evaluations. This can be observed in the reconstructed images depicted in Fig. 6.12.

The two **AEs** depicted in Fig. 6.13 reinforce our earlier observations regarding the adverse impact of excessive Convolution layers on performance, as well as the favorable outcomes associated with a higher ratio of Batch Normalization to Convolution layers. Simplified architectures appear to be particularly effective for such datasets.

Table 6.9 Convolutional **AE** results for experiments 1–5. x and y units refer to the number of filters for the convolution layers in the encoder, CIFAR-100 (Fine) dataset. **Time showed is the average for every generation, for the whole population of individuals.** AE—AutoEncoder

Gen	AE	Units (x-y)	Fitness (↓)	FID (↓)	PSNR (↑)	SSIM (↑)	MSE (↓)	Time (↓)
400	1	16–64	0.415	589.115	59.649	0.301	0.070	15 min 15 s
400	2	16–64	0.371	586.226	69.365	0.782	0.007	15 min 27 s
400	3	16–64	0.424	587.721	58.324	0.358	0.095	14 min 39 s
400	4	16–64	0.471	585.006	53.984	0.025	0.259	16 min 06 s
400	5	16–64	0.454	584.028	53.385	0.013	0.298	16 min 37 s

Table 6.10 Average results from Table 6.9 containing Convolutional **AE** results for experiments 1–5, CIFAR-100 (Fine) dataset. **Time showed is the average for every generation, for the whole population of individuals.** AE—AutoEncoder

Gen	Units (x-y)	Fitness	FID	PSNR	SSIM	MSE	Time
400	16–64	0.427	586.42	58.942	0.296	0.146	15 min 31 s

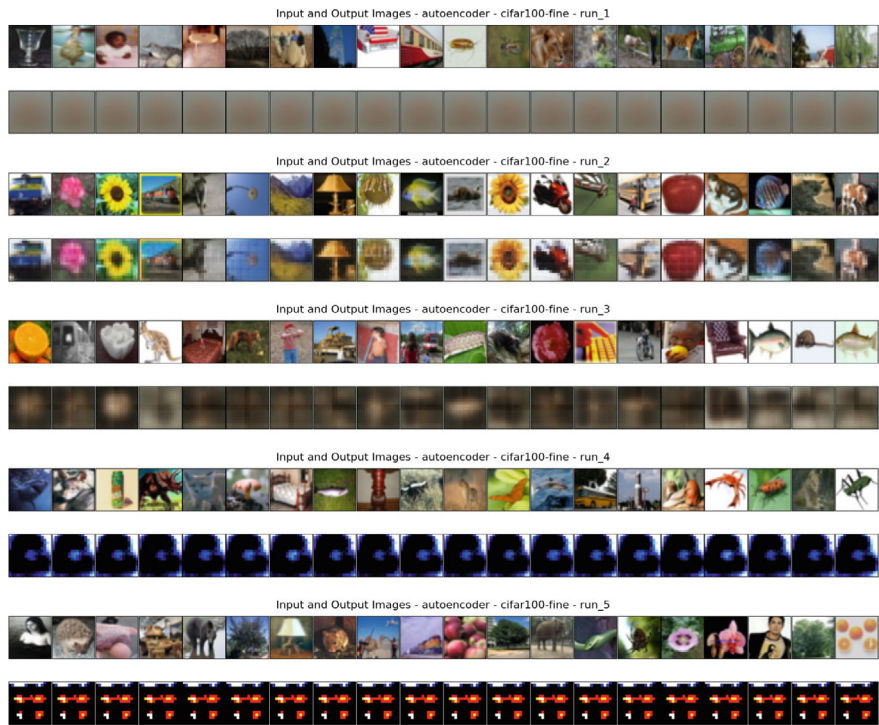


Fig. 6.12 Convolutional AE—Experiments 1–5—Input and output images, CIFAR-100 (Fine) dataset

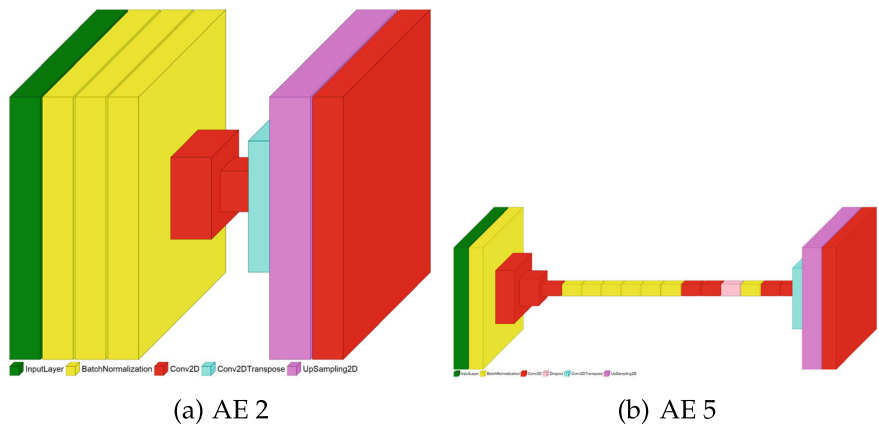


Fig. 6.13 Convolutional AE—AE architectures, Experiments 2 and 5, CIFAR-100 (Fine) dataset.
Note 1 The scale is different in each image. *Note 2* Each color represents a layer; Green: Input—Yellow—Batch normalization; Red—Convolution; Pink—Dropout; Blue—Convolution transposed; Magenta—UpSampling

Table 6.11 Convolutional AE results for experiments 6–15. x and y units refer to the number of filters for the convolution layers in the encoder, CIFAR-100 (Fine) dataset. **Time showed is the average for every generation, for the whole population of individuals.** AE—AutoEncoder

Gen	AE	Units (x-y)	Fitness (↓)	PSNR (↑)	SSIM (↑)	MSE (↓)	Time (↓)
400	6	16–64	0.101	69.871	0.823	0.006	07 min 13 s
400	7	16–64	0.062	80.994	0.985	5.17E-04	07 min 39 s
400	8	16–64	0.111	67.493	0.605	0.011	06 min 30 s
400	9	16–64	0.117	66.152	0.536	0.015	09 min 30 s
400	10	16–64	0.111	67.448	0.626	0.011	09 min 00 s
400	11	16–64	0.093	72.193	0.937	0.003	05 m 56 s
400	12	16–64	0.086	74.034	0.966	0.002	12 min 32 s
400	13	16-64	0.078	76.403	0.956	0.001	11 min 52 s
400	14	16-64	0.117	66.0969	0.565	0.015	10 min 55 s
400	15	16-64	0.106	68.728	0.907	0.008	10 min 10 s

Table 6.12 Average results from Table 6.11 containing Convolutional AE results for experiments 6–15, CIFAR-100 (Fine) dataset. **Time showed is the average for every generation, for the whole population of individuals.** AE—AutoEncoder

Gen	Units (x-y)	Fitness	PSNR	SSIM	MSE	Time
400	16–64	0.096	71.491	0.867	0.007	10 min 12 s

To further test the inefficiency of the FID score, we ran an additional set of ten experiments with the same experimental setup but using custom fitness as the fitness metric and excluding the FID score in our quality assessment metrics.

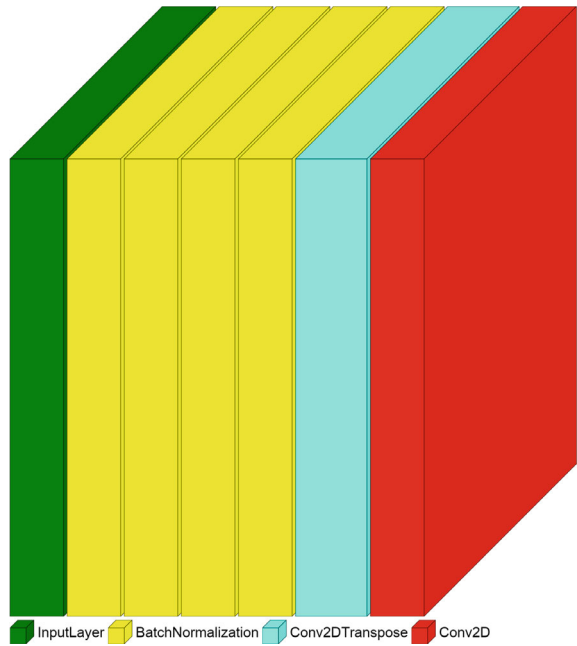
Model 7 from Table 6.11 demonstrates superior results across all metrics except for training time. In comparison to Table 6.10, Table 6.12 highlights the effectiveness of the custom fitness function over the FID score. Every value is significantly improved, corroborating our earlier findings about the FID in evaluating the quality of reconstructed images.

As displayed in Fig. 6.14, the reconstructed output quality from the 7th model aligns with Table 6.11 results, making it the best model found so far. Its architecture is displayed in Fig. 6.15, made of only three Batch Normalization layers found by the evolutionary process.



Fig. 6.14 Convolutional AE - Experiment 7—Input and output images, CIFAR-100 (Fine) dataset

Fig. 6.15 Convolutional AE—AE architecture, Experiment 7, CIFAR-100(Fine) dataset.
Note Each color represents a layer; Green: Input—Yellow—Batch normalization; Red—Convolution; Pink—Dropout; Blue—Convolution transposed; Magenta—UpSampling



6.5 Conclusions and Future Work

This work outlines the adaptations made to the **Fast-DENSER** framework to allow the automatic design and evolution of **AEs**. It demonstrates how these adaptations enable the isolation and evolution of the encoder and decoder components of an **AE**, independently observing the resulting architecture’s impact on the component’s counterpart.

Table 6.13 showcases the average results from every experiment for the three types of architectures, using the same custom fitness function. The experiments made with the **FID** score as a fitness function were excluded, as we concluded that it proved inefficient and greatly degraded the quality of the experiments.

The Fixed-Decoder architectures yield the most interesting results, namely due to the larger variety of experiments and modifications made, followed by the Convolutional and Fixed-Encoder architectures. Although the results between the three types of architectures seem marginally close, the Fixed-Decoder architectures dis-

Table 6.13 Average results from Table 6.9 containing Convolutional AE results for experiments 1–5, CIFAR-100 (Fine) dataset. **Time showed is the average for every generation, for the whole population of individuals.** AE—AutoEncoder

Arch	Units (x-y)	Fitness	PSNR	SSIM	MSE	Time
FD	Variable	0.100	71.318	0.848	0.016	08 min 42 s
FE	Variable	0.115	66.880	0.661	0.017	09 min 10 s
CAE	16–64	0.099	70.942	0.791	0.008	10 min 12 s

play faster training and higher **SSIM**, which can be more useful for a wider variety of images while also being computationally efficient. The Fixed-Encoder architectures yield the worst results, most of the time encountering difficulties in resolving blurriness issues observed in some models. This reinforces the need to better tailor Fixed-Decoder architectures.

Moreover, these architectures reveal a correlation between image reconstruction quality and the balanced ratio of Batch Normalization to Convolution layers. Simplified architectures tend to outperform more complex ones for these datasets, while an excessive number of Convolution layers can worsen the model performance. Additionally, two challenges emerged, such as: (i) the number of filters in Convolution layers needing to be tailored to the dataset’s complexity; and (ii) achieving a balance in Dropout layers to mitigate overfitting while preserving reconstruction quality. We also observed that evolution tends to favor architectures characterized by a higher prevalence of Batch Normalization layers.

For future research, we intend to explore alternative fitness metrics, while maintaining **MSE**, **PSNR**, and **SSIM** as primary metrics for assessing fitness quality. More experiments also need to be done on Fixed-Encoder and Convolutional architectures, given the preliminary stage of the conducted experiments and the lack of dataset diversity for the Convolutional experiments. Finally, we intend to switch the focus toward developing denoising **AEs** and evaluating their performance against the results showcased in this work. Furthermore, exploring the utilization of more complex datasets and adapting architectures and layer hyper-parameters to suit these datasets are also interesting aspects to investigate. It would also be interesting to explore and analyze another early stop mechanism related to the number of generations instead of the number of epochs, halting the program if the training for more generations would not prove to be beneficial.

Acknowledgements This work was supported by Project No. 7059—Neuraspace—AI fights Space Debris, reference C644877546-00000020, supported by the RRP—Recovery and Resilience Plan and the European Next Generation EU Funds, following Notice No. 02/C05-i01/2022, Component 5—Capitalization and Business Innovation—Mobilizing Agendas for Business Innovation, and within the scope of CISUC R&D Unit—UIDB/00326/2020.

References

1. Ahn, H., Yim, C.: Convolutional neural networks using skip connections with layer groups for super-resolution image reconstruction based on deep learning. *Appl. Sci.* **10**(6), 1959 (2020). <https://doi.org/10.3390/app10061959>, <https://www.mdpi.com/2076-3417/10/6/1959>
2. Assunção, F., Lourenço, N., Machado, P., Ribeiro, B.: DENSER: deep evolutionary network structured representation. *Genet. Program. Evol. Mach.* **20**(1), 5–35 (2019). <https://doi.org/10.1007/s10710-018-9339-y>
3. Assunção, F., Lourenço, N., Machado, P., Ribeiro, B.: Fast-DENSER++: evolving fully-trained deep artificial neural networks (2019)
4. Assunção, F., Lourenço, N., Ribeiro, B., Machado, P.: Fast-DENSER: fast deep evolutionary network structured representation. *SoftwareX* **14**, 100,694 (2021). <https://doi.org/10.1016/j.softx.2021.100694>, <https://linkinghub.elsevier.com/retrieve/pii/S235271102100039X>
5. Dong, C., Loy, C.C., He, K., Tang, X.: Image super-resolution using deep convolutional networks (2015). *ArXiv:1501.00092* [cs]
6. Gondara, L.: Medical image denoising using convolutional denoising autoencoders (2016)
7. Hashim, A.J., Balafar, M.A., Tanha, J., Baradarani, A.: AEVAE: adaptive evolutionary autoencoder for anomaly detection in time series. *IEEE Trans. Neural Netw. Learn. Syst.* 1–12 (2023). <https://doi.org/10.1109/TNNLS.2023.3337243>, <https://ieeexplore.ieee.org/document/10345653>. Conference Name: IEEE Transactions on Neural Networks and Learning Systems
8. Kulkarni, U., Patil, S., Vikas, E., Patil, R., Kulkarni, B., Meena, S.M.: Image denoising using autoencoders : denoising noisy images by removing noisy pixels/grains from natural images using deep learning and autoencoders techniques. In: 2023 IEEE 8th International Conference for Convergence in Technology (I2CT), pp. 1–6. IEEE, Lonavla, India (2023). <https://doi.org/10.1109/I2CT57861.2023.10126382>, <https://ieeexplore.ieee.org/document/10126382/>
9. Liang, J., Liu, R.: Stacked denoising autoencoder and dropout together to prevent overfitting in deep neural network. In: 2015 8th International Congress on Image and Signal Processing (CISP), pp. 697–701. IEEE, Shenyang, China (2015). <https://doi.org/10.1109/CISP.2015.7407967>, <http://ieeexplore.ieee.org/document/7407967/>
10. Mishra, S., Jabin, S.: Anomaly detection in surveillance videos using deep autoencoder. *Int. J. Inform. Technol.* (2023). <https://doi.org/10.1007/s41870-023-01659-z>
11. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting
12. Suganuma, M., Ozay, M., Okatani, T.: Exploiting the potential of standard convolutional autoencoders for image restoration by evolutionary search. In: Proceedings of the 35th International Conference on Machine Learning, pp. 4771–4780. PMLR (2018). <https://proceedings.mlr.press/v80/suganuma18a.html>. ISSN: 2640-3498
13. Wang, Z., Bovik, A., Sheikh, H., Simoncelli, E.: Image quality assessment: from error visibility to structural similarity. *IEEE Trans. Image Process.* **13**(4), 600–612 (2004). <https://doi.org/10.1109/TIP.2003.819861>, <https://ieeexplore.ieee.org/document/1284395>. Conference Name: IEEE Transactions on Image Processing
14. Yeh, R.A., Chen, C., Lim, T.Y., Schwing, A.G., Hasegawa-Johnson, M., Do, M.N.: Semantic image inpainting with deep generative models (2017). *ArXiv:1607.07539* [cs]

Chapter 7

Code Building Genetic Programming is Faster than PushGP



Edward Pantridge and Thomas Helmuth

Abstract Users of general program synthesis tools aim to increase their productivity as a programmer by automating the creation of simple programs. Although there are many contemporary program synthesis methods with different trade-offs, Genetic Programming (GP) has remained an appealing choice when performing inductive program synthesis from a dataset of training cases. One common critique of GP is that the time required to evolve a solution program is often considered prohibitively long to help with programmer productivity. In this chapter, we present empirical measurements of the runtime costs of the relatively new GP method Code Building Genetic Programming (CBGP) compared to PushGP and show an order of magnitude reduction in runtime. We also discuss general methods for reducing the runtime of GP systems that are especially well suited for CBGP implementations.

7.1 Introduction

The field of general program synthesis has been receiving increased attention in recent years due to the introduction generative *large language models* (LLMs) that can produce executable programs in a variety of programming languages based on a user-supplied “prompt” that describes the target program using natural language. These tools have seen widespread adoption across many practitioners who develop and maintain software professionally. Although the correctness and quality of synthesized programs produced by generative LLMs are often impressive in their own right, users frequently claim the short response time in which they receive a synthesized program as the primary reason they find these tools useful. In other words, the increase in productivity comes from getting a correct, or mostly correct, program quickly for the user to adjust as needed.

E. Pantridge (✉)
Spindle Health, Boston, MA, USA
e-mail: eddie@spindlehealth.com

T. Helmuth
Computer Science, Hamilton College, Clinton, NY, USA
e-mail: thelmuth@hamilton.edu

Despite their widespread use and media attention, generative LLMs are not a universal solution for program synthesis. To create (i.e., train) a new LLM, a practitioner must have access to a large corpus of high-quality source code for programs that are associated with similarly high-quality descriptions. Using and fine-tuning pre-trained LLMs is possible without such a corpus, but the quality of synthesized programs with respect to correctness generally suffers in scenarios where the user's target program differs from programs found in the original training corpus. In addition, the LLMs with the best performance across multiple problem domains tend to require specialized infrastructure that comes with significant computational and financial costs. For this reason, most users delegate the responsibility of hosting an LLM for program synthesis to a third party and often sacrifice the ability to customize the LLM to their problem domain.

Genetic Programming (GP) is a family of program synthesis methods that excel at *inductive* program synthesis. In contrast with LLMs, GP systems primarily leverage a dataset of example input-output pairs to specify the program that the system should generate. Generally speaking, GP systems don't require a corpus of source code or natural language, nor any up-front training computation. The use of GP doesn't necessitate specialized hardware architectures, although one can be used when synthesizing programs that benefit from such hardware.

Although GP is a suitable program synthesis solution in some scenarios it has not seen the same adoption as LLMs. There are many possible explanations for this phenomenon. Practitioners may prefer a natural language interface over inductive synthesis. LLMs may be more likely to produce a correct or easier to understand than program compared to GP. Perhaps most importantly, the time it takes to synthesize a program with GP is too long to be useful in most real-world settings.

Much of the GP research literature for program synthesis has focused on improving the solution finding capabilities of various GP methods across a wide range of problem domains. In this chapter, we present evidence and explanation for how a recently developed GP method, *Code Building Generic Programming* (CBGP), has dramatically reduced the computational cost associated with evolving programs. Much like LLMs, this reduction in runtime likely makes CBGP more appealing to practitioners compared to other GP methods, but also may allow users with large computational budgets to run evolution with significantly larger populations or for longer evolutionary time in order to solve more difficult problems.

We begin the rest of this chapter with a summary of the two contemporary genetic programming methods that we will compare in our experiments. We then present some cost measurements taken from both GP systems and a discussion on why certain aspects of each method lead to different computational costs. Finally, we discuss evidence for specific future improvements to CBGP that could be used to reduce computational cost even further.

7.2 PushGP

Push is a programming language developed to be the language in which evolving programs are run in a GP system [18]. Push was designed to emphasize evolvability, simplicity of syntax, and uniformity. Any nested list of Push instructions and literals is a valid program that will execute without runtime errors. This simple syntax makes it easy to generate random programs and vary them through genetic operators while ensuring code that will run. Push programs can manipulate multiple data types and use a wide-ranging set of control structures, making them capable of use for general-purpose programming.

In Push, the state of a running program is maintained on a set of typed stacks, such as an integer stack, a Boolean stack, and a string stack. Additionally, the state contains an exec stack, which controls the execution of the program. At the start of execution, every element of the program is placed on the exec stack, with the start of the program at the top of the stack. A program is executed by repeatedly popping the top instruction, literal, or nested list from the top of the stack and evaluating it.

Each Push instruction takes its arguments from the appropriate stacks and returns its result to a stack. For example, the `integer_gt` instruction pops two integers from the integer stack, determines if the second popped is greater than the first, and pushes the result onto the Boolean stack. Literals are simply pushed onto their relevant stacks. When a nested list is on top of the exec stack, each of its elements is pushed onto the exec stack.

Push programs are typically interpreted by an interpreter written in a host language. The interpreter repeatedly executes the top element on the exec stack until either the exec stack is empty, or the interpreter reaches some predetermined number of execution steps. Push implements control structures using a variety of instructions that manipulate the exec stack while the program is running. Push has exec stack instructions implementing everything from standard if/else statements and loops to iterators to more exotic control structures. Whether a program terminates with an empty exec stack or by an execution limit, the program takes an element from the top of one of the stacks as the return value of the program. For example, if the problem at hand requires the program to return a string, it will be taken from the top of the string stack.

One overarching principle of Push is that in situations when an instruction would cause a runtime error, instead, it “NOOPs,” i.e., it has no effect on the state of the stacks. A common example occurs when an instruction requires more arguments of a type than can be found on that type’s stack; perhaps there is only one integer on the integer stack when executing `integer_gt`. In such cases, none of the potential arguments are popped off of their stacks. As another example, the `vector_integer_first` instruction takes the top vector of integers off of the vector of integers stack and pushes its first element onto the integer stack. However, if the top vector is empty, there is no first integer, so `vector_integer_first` instead NOOPs.

7.3 Code Building Genetic Programming

Similar to PushGP, *Code Building Genetic Programming* (CBGP) is a genetic programming method that evolves programs by searching over the space of variable length linear genomes. Unlike PushGP, CBGP constructs a type-safe abstract syntax tree (AST) in the host language from each genome and delegates the execution of the program to the runtime environment of host language [16]. This process is done using a stack of ASTs and a formal type system for composing ASTs such that the composition is guaranteed to be type safe [15].

Figure 7.1 shows the state of the AST stack during each step of compilation for a simple genome. Also shown is the type environment that provides the data type for all variables, including program inputs and global functions. The APP gene causes CBGP to search the stack from top to bottom to find an AST whose type is a function. If found, the stack will be searched for other ASTs to use as arguments. The type environment and type unification system are used to ensure that valid argument ASTs are chosen so that the composite AST is type safe and has a known output type. When using a sufficiently sophisticated type unification system, CBGP can evolve programs which use polymorphic data types and functions [14].

CBGP also supports function abstraction (ABS) genes which create new functions within the body of the AST and let binding (LET) genes for assigning immutable local variables. The compilation process for these gene types requires the addition of new local variables in the type environment. References to these new local variables are created when processing a local variable gene (LOCAL[i]) that is annotated with a random index (i) that is used to select one of the local variables in scope using modulo logic.

The compilation process can NOOP when an APP gene cannot find a function AST to call or an argument AST to pass to the function. A LET gene can noop if the

Genome:				Var	Type
[in1, len, APP, 2, *, APP]				in1	String
				len	String → Int
				*	(Int,Int) → Int
				*	
	len	2	2		
<u>in1</u>	<u>in1</u>	<u>len(in1)</u>	<u>len(in1)</u>	<u>len(in1)</u>	<u>len(in1)*2</u>
Step 1	Step 2	Step 3	Step 4	Step 5	Step 6

Fig. 7.1 A demonstration of the compilation process that converts a simple linear genome into a type-safe program. For each step of compilation we show the state of the AST stack. Complex expressions are shown as source code for clarity, but in implementation all elements of the stack are trees. The type environment that provides the data types of every variable is also shown

AST stack is empty. All literal and variable genes are pushed to the AST stack and are therefore never a NOOP. After all genes have been processed there will be zero or more ASTs on the stack. The top-most AST that returns the data type of the target program is selected as the program associated with the genome. All other ASTs on the stack are discarded.

The final AST is passed to the runtime environment of the host language to create a function object in the same representation as all other functions provided by the host language. In most cases, the AST is given to the same compiler used by human programmers to produce executable machine code. The CBGP implementation used to gather the empirical results of this paper is hosted on the Java Virtual Machine (JVM) and the ASTs produced for each genome are compiled into JVM bytecode objects that implement a function-like interface that allows them to be called on arguments. This compilation happens at runtime during evolution and the resulting bytecode is injected into the same process where the genetic algorithm is running.

When evolution finds a solution program, the CBGP system must output the program in a form usable by the user. The AST is a natural representation for both humans and systems to manipulate. It is also possible to translate the AST into source code, depending on the host language.

Previous research into the solution finding capabilities for PushGP and CBGP have shown that PushGP generally has a higher solution rate on the *Program Synthesis Benchmark Suite* versions 1 and 2. On problems where CBGP is able to find solutions, it almost always finds the solution more consistently than PushGP [15].

7.4 Experimental Design

CBGP and PushGP have many similarities. They both search for variable length linear genomes that correspond to a solution program. The “genes” within those genomes are atomic units of programs (e.g., literals, variables, functions, etc.). Stack-like data structures are used to preserve type safety. The primary distinction between CBGP and PushGP is the pre-compilation of ASTs in CBGP versus the repeated use of the stack-machine interpreter in PushGP systems. Other differences have been discussed in prior work, including the empirically observed solution rates on benchmark problems and support for polymorphic types and functions. The impact that this pre-compilation has on the computational cost has been under reported, until now.

We present empirical measurements of the CPU time taken to evaluate a population of 1000 genomes using CBGP vs PushGP across 3 benchmark problems. Each measurement corresponds to a generation seen during one of 10 full evolutionary runs performed using each GP system on every problem.

Both GP implementations are hosted on the JVM and implemented in the Clojure programming language, but one uses CBGP and the other uses PushGP. The function sets provided to each GP implementation match as closely as possible to each other for each benchmark problem. Each run was performed in an isolated process that was

given a single core of a 2.6 GHz CPU and 2GBs of RAM. No multi-threading capabilities were used. The same genome sizes, selection methods, and variation operators were used in all runs; however, the CPU time taken for selection, variation, and any aspect of evolution aside from evaluation is not included in these measurements.

The benchmark problems were chosen from the program synthesis benchmark suite version 1 [8]. Descriptions of the target solution programs are as follows:

Vectors Summed: Given two equal-sized vectors of integers, return a vector of integers that contains the sum of the input vectors at each index.

Negative To Zero: Given a vector of integers, return the vector where all negative integers have been replaced by 0.

Replace Space With Newline: Given a string input, print the string, replacing spaces with newlines. Also, return the integer count of the non-whitespace characters. The input string will not have tabs or newlines.

The *vectors summed* and *negative to zero* problems involve manipulation of vector data structures containing integers while *replace space with newline* requires transforming a string of characters and printing to a buffer. All problems require at least one form of control flow, specifically iteration. The problems selected for this study were chosen because the solution rates of both CBGP and PushGP are known to be relatively high. This allowed us to take measurements of the program evaluation costs throughout a run in order to sample populations of randomly generated genomes and evolved populations that are close to the target solution.

Both GP implementations were driven by a generational evolutionary algorithm for a maximum of 300 generations. Evolution stopped prior to the maximum number of generations if a program in the evolved population passed all training cases. The genetic operators and other hyperparameters for both PushGP and CBGP are given in Table 7.1.

Table 7.1 The evolutionary hyperparameters used for the PushGP and CBGP runs presented in this work

Hyperparameter	Value
Population size	1000
Max generations	300
Parent selection	Lexicase selection [9]
Variation	UMAD [7]
Mutation rate	0.1
Initial genome sizes	[50, 250]
Number of training cases	200
Runs per problem	10

7.5 Cost of Evaluations

Figure 7.2 shows the distribution of CPU time measurements for each GP system and benchmark problem. It is clear that the cost of evaluation is dramatically lower using CBGP.

In the most extreme of the three comparisons, the mean runtime for evaluating a generation of 1000 individuals during a run of “vectors summed” using PushGP is approximately 13.5 min. To perform the same number of evaluations using CBGP requires an average of only 8 s. This means that CBGP has computational savings of over 99% on this problem. While the other two problems exhibit a speedup of roughly an order of magnitude.

Many of the individual generation measurements summarized in Fig. 7.2 are taken from the same runs, and thus are not independent due to the lineage of the population. Figure 7.3 shows the evaluation times throughout evolutionary time. The overall CPU time comparison from Fig. 7.2 still holds. Figure 7.3 also shows that PushGP tends to exhibit a lot more variance in evaluation time compared to CBGP.

In the rest of this section, we will describe the specific mechanisms that we hypothesize explain this difference in computational cost.

7.5.1 Indirect Encoding to Machine Code

The CBGP compilation procedure produces an AST in the host language which a subsequent compiler translates into machine code. Performing this process once per individual has some added cost; however, the machine code representation dramatically reduces the overhead of each function invocation on the training dataset. The

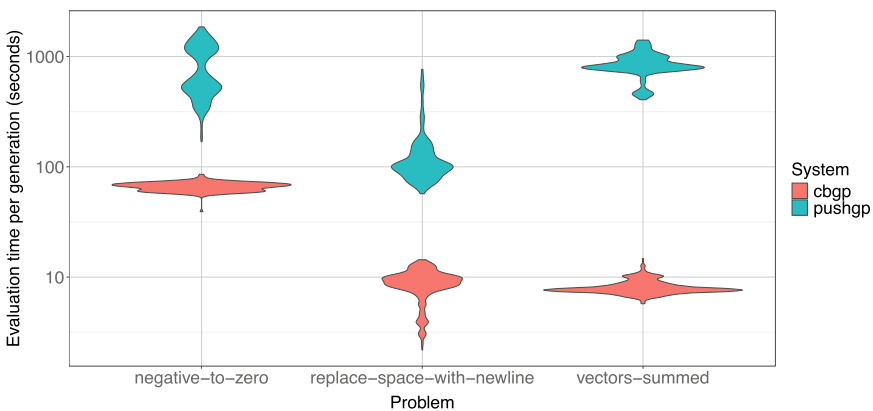


Fig. 7.2 The distribution of CPU time spent evaluating one generation of 1000 individuals. Note the log scale on the vertical axis

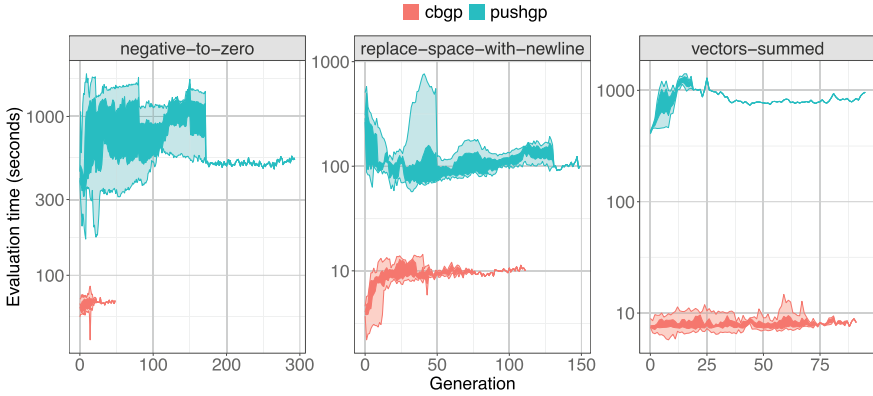


Fig. 7.3 The distribution in CPU time spent evaluating each generation of 1000 individuals across evolutionary time. Note the log scale on the vertical axis

compiler of the host language often produces machine code that is much smaller in terms of memory and better utilizes CPU caches among other optimizations. This faster machine code is run many times, once per training case, compared to the single (slower) compilation step, which only needs to run once per individual.

PushGP systems require an interpreter of the Push language to be implemented in the host language. This interpreter controls all data flow between functions (a.k.a. instructions) using stacks. In other words, the result of each computation is always pushed to a stack and must be popped by a subsequent computation that will use the value as an input. This manipulation of stack structures adds significant overhead. More complex computations must manage multiple intermediate values that temporarily live on stacks while further steps of the interpreter are performed. For example, an iteration instruction in the Push language might repeatedly pop a positive integer to serve as a loop counter, decrement it, and push it back to the stack, while performing other instructions that make up the “body” of the loop until the integer reaches zero. This superfluous movement of data and updating of references takes CPU time and also makes it much less likely that commonly used pieces of data, such as the loop counter, will be in the CPU caches on subsequent retrievals.

Figure 7.4 shows a simplified trace of CPU operations for executing a simple PushGP genome. It includes operations that push and pop values from an exec and integer stacks. It does not include CPU operations for checking if stacks are empty, counting interpreter steps, or checking interpreter limits. In this crude example, only 12.5% of the CPU operations correspond to the logical semantics of the program.

The Push interpreter must be run in full for every execution of the program on a training case. While the interpretation of a Push program may be approximately as costly as compiling a CBGP genome into its AST, the latter only needs to run once per individual, while the former runs many times. We hypothesize that this difference plays a major role in the observed differences in computational time.

Fig. 7.4 A simplified trace of CPU operations for executing a PushGP genome of length 5 which multiplies an input value by 2 and then divides the product by the input value, leaving a 2 on the integer stack. The two steps in boxes correspond to operations related to program semantics, and the remaining 14 steps are simply moving values to and from the stacks

Genome:	[in1, 2, int_mult, in1, int_div]
	1. POP exec
	2. PUSH in1 → Int
	3. POP exec
	4. PUSH 2 → Int
	5. POP exec
	6. r1 ← POP Int
	7. r2 ← POP Int
	8. r3 ← r1 * r2
	9. PUSH r3 → Int
	10. POP exec
	11. PUSH in1 → Int
	12. POP exec
	13. r1 ← POP Int
	14. r2 ← POP Int
	15. r3 ← r1 / r2
	16. PUSH r3 → Int

7.5.2 Dead Gene Elimination

Dead code elimination refers to a commonly used compiler optimization that removes code which provably does not affect the program’s behavior. Such code is called “dead” as it either is unreachable or the result is never used. Examples of dead code include assigning variables that are not referenced, conditional branches whose conditional cannot be satisfied, and any logic following a `return` or other statement that causes execution to leave the current scope. Removing this code helps reduce resource utilization by shrinking the size of the program and eliminates the possibility of executing superfluous operations.

Similar to dead code, the variable length linear genomes evolved by both CBGP and PushGP contain “dead” genes that do not impact the behavior (or structure, in the case of CBGP) of the synthesized program.

In Sects. 7.2 and 7.3, the conditions under which a gene is considered a NOOP are described for PushGP and CBGP, respectively. In PushGP, the execution of the program may experience a NOOP of different genes depending on inputs passed to the program due to conditionals and other forms of control flow. This makes it difficult to know which genes will always NOOP or if a gene that has been shown to NOOP under some inputs is safe to be considered dead. In contrast, the compilation of an AST in CBGP is completely determined by the genome alone. This means that any gene that does not contribute to the final AST is proven to be a dead gene.

Both PushGP and CBGP genomes can have dead genes that are not NOOPs. Figure 7.5 shows an example PushGP genome that contains genes which do not

Genome:	[<u>1</u> , "A", <u>2</u> , <u>int_add</u> , true, bool_and, "Z"]									
			"Z"							
Final Stacks:	3	"A"	true					Output:	3	
	Int	String	Boolean							

Fig. 7.5 An example PushGP genome and the final state of all stacks after running the program through an interpreter. Assuming the problem specifies an integer output, the program output will be the top integer (in this case, 3). The underlined genes are the only genes that are not considered “dead” because they influence the program’s output. The remaining genes will either always NOOP during execution (e.g., `bool_and` has insufficient arguments) or will only affect a stack other than the integer stack

NOOP but are still considered dead because they don’t change the output value produced by the program. This often occurs when a gene modifies a stack other than the stack from which the output value is retrieved. However, it is not possible to prove that a PushGP gene is dead simply by looking at the stacks it modifies because the program may contain a later gene that uses a value from one stack to modify another. For example, the `string_length` instruction pushes a value to the *Int* stack corresponding to the length of the top value on the *String* stack, and therefore, if a problem outputs an *Int* value it is not generally true that genes which modify the *String* stack are dead. Furthermore, most Push instructions consume their inputs (aka pop them from the stack) which makes them unavailable to subsequent instructions. This means that an instruction may influence the output of the program even if the values produced by that instruction are not used by subsequent instructions.

CBGP genomes can also include dead genes that are not NOOPs during compilation of ASTs. Figure 7.6 provides a CBGP genome and the final state of the AST stack. From the final stack, the top-most AST with the correct output type for the problem is chosen as the genome’s corresponding program. All genes that impacted the creation of any AST other than the chosen AST are dead. For this reason, we

Genome:	[<u>1</u> , "A", <u>2</u> , <u>+</u> , true, <u>APP</u> , <u>&</u> , APP, "Z"]									
		"Z"	String							
		2+1	Int							
Final Stack:	true	Boolean						AST:	2+1	
	"A"	String								
	AST	Type								

Fig. 7.6 An example CBGP genome and the final state of the AST stack after genome compilation. Assuming the problem specifies an integer output, the top-most integer AST will be chosen as the genome’s program (in this case $2 + 1$). The underlined genes are the only genes that are not considered “dead” because they are present in the final AST. The remaining genes will either NOOP during compilation or appear in an AST other than the one that is chosen for evaluation

say that CBGP provides implicit “dead gene elimination” because the compilation process happens once per genome prior to any program executions on the training data and therefore the dead genes have a negligible computational cost.

The crucial distinction between CBGP and PushGP with respect to dead genes is that PushGP processes these genes once per program execution while CBGP processes these genes once per individual. This means that any operation involved in the processing of dead genes in a PushGP genome will be repeated on the order of the size of the training dataset. In other words, supplying more training data to PushGP system also increases the amount of computational resources wasted on dead genes. CBGP saves this cost with a one-time compilation of the genome into an AST that does not reference any dead genes.

Under both GP methods, the dead genes remain in the genome representation that is used for recombination. These dead genes could be important for allowing evolution to carry dead genes across generations only to later produce a child where this genetic material is no longer dead. This *genetic neutrality* has been shown to be useful for GP in order to move through neutral parts of the search space in order to find better individuals [3, 11, 12, 19]. Thus, we do not make any effort to remove dead genes from either PushGP or CBGP genomes as part of the evolutionary process.

7.5.3 Auto-Simplification

The GP literature contains many methods for automatically simplifying programs. For PushGP specifically, it has been shown that a stochastic hill-climbing method of simplification can be used to dramatically reduce the size of evolved programs and improve their generalization to unseen test cases [4]. Under the scheme, a random subset of genes is removed from the genome and it is sent through the entire evaluation procedure to check if the semantics of the program changed. Although this process typically removes most, if not all, of the dead genes, it is also an expensive process that is typically only applied to a single solution program at the end of the evolutionary run prior to assessing generalization on an unseen test set. CBGP has the advantage that most of the dead genes are not included in the AST based solely on way the AST is constructed and no additional program evaluations are needed for simplification.

7.6 Caching Evaluations by Semantics

In this section, we speculate about future developments that could further improve the running time of CBGP. All methods are theoretically applicable to other GP methods; however, we present some practical benefits of implementing these improvements in a CBGP system.

Programs with different syntax can have the same *semantics*. In this chapter, we define a program’s semantics as the relationship between inputs and outputs produced

Algorithm 7.1 Different functions in the same equivalence class

```

function JUSTFOUR
  return 4
end function

function FOURFROMSTRING
  return LENGTH("four")
end function

function SLOWFOUR
   $i \leftarrow 0$ 
   $four \leftarrow 4$ 
  while  $i < 1000000$  do
     $four \leftarrow four + 0$ 
     $i \leftarrow i + 1$ 
  end while
  return  $four$ 
end function

```

by a program. In other words, two deterministic programs with the same semantics will always produce the same output when provided the same inputs.

The set of programs that have identical semantics are said to belong to the same *equivalence class*. By definition, all programs within an equivalence class will be assigned the same error (i.e., fitness) under a GP system that use a static set of training cases to guide evolution. For many equivalence classes, there are an infinite number of program structures that are members of the class. For example, the three functions in Algorithm 7.1 all take no arguments and always return the integer 4. Clearly these functions will produce identical outputs under all conditions despite each having a different syntax.

If it were possible to analyze the structure of a program during evolution and prove that a program from the same equivalence class had already been evaluated, the GP system could skip evaluation of the program and assign it the errors (or fitness) of the previously seen program.

It should be noted that members of an equivalence class do not necessarily have the same computational cost. If a program's ASTs could automatically be rewritten into a different AST from the same equivalence class that is cheaper to compute, there may be a net reduction in cost when evaluating the program on a large training dataset. This is an interesting area of research, but distinct from the improvement proposed in this section. Instead, we solely focus on association of programs to a previously seen equivalence class with the intention of skipping redundant evaluations.

In the remainder of this section, we will motivate and describe a strategy for caching programs according to their semantics in order to eliminate program evaluations for programs in an equivalence class seen previously during evolution. We begin by showing measurements that suggest potential for significant cost savings using this approach and end with a discussion of how such a system could be implemented in CBGP.

7.6.1 Semantic Equivalence of CBGP Programs

For the purposes of this discussion, we define a program’s *behavior* differently than its semantics. The remainder of this work will use the term behavior to mean the set of outputs produced by a program on a specific set of inputs (aka cases). Programs with the same semantics are guaranteed to have the same behavior; however, programs with the same behavior are not guaranteed to have the same semantics. Despite this, behavioral equality between programs can be used to predict semantic equality if the set of cases used to derive behavior is sufficiently representative of program’s full domain.

To estimate the potential computational savings from evaluating each equivalence class only once, we measured the cardinality of program behaviors, $|B|$, seen across an entire CBGP run and compared it to total number of individuals, $|I|$, produced throughout the run. Assuming behavior equality is an accurate predictor of semantic equality, this gives us the proportion of evaluations that corresponded to novel equivalence classes according to our training cases. Equivalently, we can derive the rate of evaluation events for programs that were in the same equivalence class as a previous seen program as given by $r = (1 - |B|/|I|)$. This rate, r , is the maximum proportion of evaluations that would be skipped if a GP system could assign programs to their equivalence class with perfect accuracy and had sufficient memory to hold all previous evaluations in a cache.

Figure 7.7 shows the distribution of r across 10 CBGP runs for each problem. The mean redundant evaluation rate for “negative to zero” is 0.93, “replace space with newline” is 0.91, and “vectors summed” is 0.71. In all problems the overwhelming majority of program evaluations were for programs with an identical behavior to another program seen before.

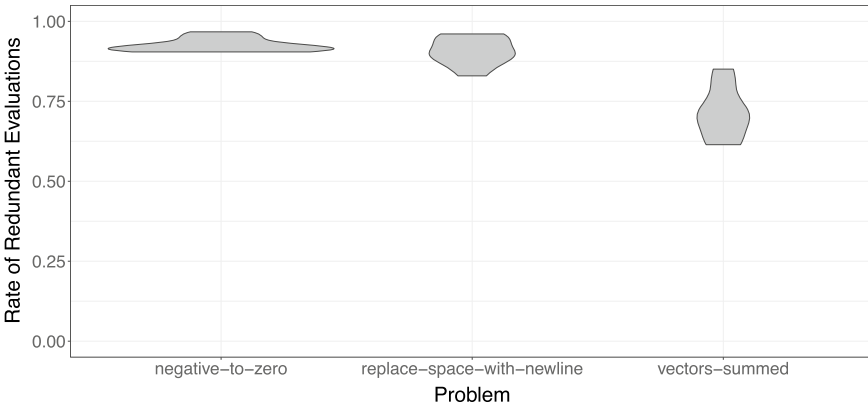


Fig. 7.7 The rate, r , of program evaluations that resulted in an identical set of outputs for the entire training dataset that had been observed from a program seen previously during the evolutionary run

This phenomenon implies the CBGP is struggling to explore new areas of program space, which may contribute to the lower solution rates on many benchmark problems compared to PushGP. This could be caused by poorly calibrated variation operators or simply be an artifact of the CBGP compilation process. Regardless, most of the computational effort being spent under the current CBGP scheme is wasted re-evaluating seemingly equivalent programs.

The measurements of r presented above are strongly related to, but distinct from, the behavioral diversity of the population, defined as the proportion of individuals with unique behaviors in a population [13]. Behavioral diversity is measured within the population of one generation of a GP run, where r is calculated based on the behaviors of individuals across all generations. Figure 7.8 shows the distribution of behavioral diversity across evolutionary time for the same 10 runs as Fig. 7.7. The diversity is consistently higher than what would be expected if the observed values of r were purely caused by low diversity in the population (as shown by the red dashed line). This shows that the high rates of r must be caused by programs from the same equivalence class occurring across generations of evolution, and thus a semantic cache would still be effective at reducing evaluations even under higher levels of behavioral diversity.

Although it is not the focus of this section, it should be noted that in previous works PushGP has been shown to achieve and maintain high levels of behavioral diversity when using lexibase parent selection [5, 6]. The drastic differences in diversity clearly point to a major difference between CBGP's and PushGP's abilities to produce a variety of semantics. Future work should explore these differences; improving diversity in CBGP could be key to unlocking its potential for better problem solving performance.

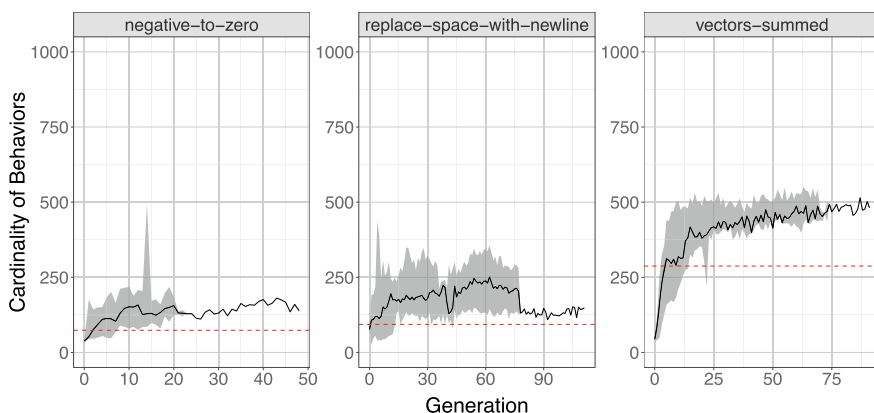


Fig. 7.8 The behavioral diversity across evolutionary time. The shaded region shows the range between the minimum and maximum diversity observed at that generation across all 10 runs and the solid line is the median. The red dashed line shows the expected cardinality of behaviors if the value of r was solely caused by behavior diversity

7.6.2 Normal Form of CGBP Programs

In order to reduce the amount of redundant evaluations, we first must detect when a program is a member of an equivalence class we have previously evaluated. To accomplish this in a way that is sound and complete for all programs is impossible because it is an undecidable problem [10, 17]. However, in the context of skipping evaluations of programs in GP the system need not be complete. In other words, a non-zero false negative rate will cause GP to perform some redundant evaluations, but there still may be cost saving proportional to the true positive rate.

Contemporary CGBP implementations use lambda calculus as the basis of the program representation. This allows for programs to be rewritten into the *normal form* by repeatedly applying three reduction rules until no further reductions can be made [1].

- α -renaming changes a bound variable name in every place it occurs within the scope.
- β -reduction replaces a function application with its output value when all arguments are known.
- η -reduction replaces function abstractions which trivially pass their arguments to another function.

These rules also have the benefit of simplifying the program's structure which could potentially reduce, but not increase, the cost of evaluating the program. The following series of reductions show that f , which is already in normal form, is in the same equivalence class as f' because f' reduces to the same normal form.

$$\begin{aligned}
 f(x) &= \text{multiply}(2, x) \\
 g(x) &= \text{double}(x) \\
 f'(y) &= \text{multiply}(g(1), y) \\
 &= \text{multiply}(\text{double}(1), y) && \text{from } \eta\text{-reduction} \\
 &= \text{multiply}(2, y) && \text{from } \beta\text{-reduction} \\
 f'(x) &= \text{multiply}(2, x) && \text{from } \alpha\text{-renaming}
 \end{aligned}$$

It should be reiterated that all lambda calculus programs have a normal form and all programs with the same normal form belong to the same equivalence class. However, there may be many different normal forms that belong to the same equivalence class. For example, the following programs f and g belong to the same equivalence class and are both in their respective normal forms.

$$\begin{aligned}
 f(x) &= \text{multiply}(2, x) \\
 g(x) &= \text{add}(x, x)
 \end{aligned}$$

For this reason, it is not guaranteed that a cache which stores program evaluations for each normal form will reduce the proportion of evaluations according to r . Instead, the observed reduction in evaluations will be some rate $r' \leq r$. In other words, there will still be some evaluations of programs from the same equivalence class. More experimentation is required to assess if CBGP repeatedly searches over programs of the same normal form.

7.6.3 Cache Eviction

As the cost of program evaluation is reduced, it is likely that practitioner will increase the population size, number of generations, or both in order to improve solution finding performance. This could result in an increase in the cardinality of normal forms seen during evolution. If the normal forms are never evicted from the cache, the resources needed to store the cache could grow to unacceptable levels. An ideal eviction policy would selectively choose the normal form to remove from the cache such that the hit rate of the cache remains high.

Evolutionary populations drift over time due to selection pressure. This means that the programs found in later generations of evolution may be significantly different than the programs in the initial random population. This suggests a *least recently used* policy would be a good choice [2]. When the cache reaches its maximum size and a novel normal form is seen, the normal form that has not been seen for the longest amount of evolutionary time is evicted. This biases the cache toward normal forms that are commonly found in the current evolutionary population.

7.7 Conclusion

In this chapter, we quantify the difference in runtime cost of CBGP compared to PushGP and describe the mechanisms which were used by CBGP to mitigate the computational overhead exhibited by PushGP. Specifically, we detail how CBGP implicitly disregards “dead” genes and pre-compiles genomes into machine code which leads to at least an order of magnitude reduction in computational cost compared to PushGP when using comparable hyperparameters on benchmark problems. Although these mechanisms are theoretically generic to other GP systems, we discuss the aspects of PushGP that make implementation nearly impossible.

We also present observations about the frequency with which CBGP spends time evaluating programs which appear to be semantically equivalent to programs evaluated previously. We propose the use of a caching strategy that allows CBGP to skip program evaluations when it can be shown that a program is in the same equivalence class as a program in the cache. Again, this technique is not unique to CBGP; however, the use of lambda calculus for the CBGP program representation allows implementations of the cache to leverage the formalism of a program’s normal form

as a sound, but incomplete, method of detecting if two programs are of the same equivalence class. Further study into the computational savings of such a cache is needed to confirm that the benefits outweigh the cost of rewriting programs into their normal form.

Acknowledgements We thank all participants of the Genetic Programming Theory and Practice XXI workshop for their discussion about this chapter. In particular, we thank the reviewers Alcides Fonseca and Erik Fredericks for their comments which greatly helped to improve this manuscript. We also thank the members of the PUSH lab at Amherst College and the University of Massachusetts, Amherst, for their input on this research. In particular, we thank Lee Spector for his continued support of this line of research.

References

1. Church, A.: A formulation of the simple theory of types. *J. Symb. Logic* **5**(2), 56–68 (1940)
2. Denning, P.J.: The working set model for program behavior. *Commun. ACM* **11**(5), 323–333 (1968). <https://doi.org/10.1145/363095.363141>
3. Galván-López, E., Poli, R., Kattan, A., O’Neill, M., Brabazon, A.: Neutrality in evolutionary algorithms. . . what do we know? *Evol. Syst.* **2**, 145–163 (2011)
4. Helmuth, T., McPhee, N.F., Pantridge, E., Spector, L.: Improving generalization of evolved programs through automatic simplification. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO’17*, p. 937–944. Association for Computing Machinery, New York, NY, USA (2017). <https://doi.org/10.1145/3071178.3071330>
5. Helmuth, T., McPhee, N.F., Spector, L.: Lexicase selection for program synthesis: a diversity analysis. In: *Genetic Programming Theory and Practice XIII, Genetic and Evolutionary Computation*. Springer, Ann Arbor, USA (2015). <http://www.springer.com/us/book/9783319342214>
6. Helmuth, T., McPhee, N.F., Spector, L.: Effects of lexicase and tournament selection on diversity recovery and maintenance. In: *GECCO’16 Companion: Proceedings of the Companion Publication of the 2016 Annual Conference on Genetic and Evolutionary Computation*, pp. 983–990. ACM, Denver, Colorado, USA (2016). <https://doi.org/10.1145/2908961.2931657>, <http://doi.acm.org/10.1145/2908961.2931657>
7. Helmuth, T., McPhee, N.F., Spector, L.: Program synthesis using uniform mutation by addition and deletion. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO’18*, pp. 1127–1134. ACM, New York, NY, USA (2018). <https://doi.org/10.1145/3205455.3205603>, <http://doi.acm.org/10.1145/3205455.3205603>
8. Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO’15*, p. 1039–1046. Association for Computing Machinery, New York, NY, USA (2015). <https://doi.org/10.1145/2739480.2754769>
9. Helmuth, T., Spector, L., Matheson, J.: Solving uncompromising problems with lexicase selection. *IEEE Trans. Evol. Comput.* **19**(5), 630–643 (2015). <https://doi.org/10.1109/TEVC.2014.2362729>, <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&number=6920034>
10. Hopcroft, J.E., Ullman, J.D.: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley (1979)
11. Hu, T., Banzhaf, W.: Neutrality and variability: two sides of evolvability in linear genetic programming. In: *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, GECCO’09*, pp. 963–970. Association for Computing Machinery, New York, NY, USA (2009). <https://doi.org/10.1145/1569901.1570033>

12. Hu, T., Banzhaf, W.: Neutrality, Robustness, and Evolvability in Genetic Programming, pp. 101–117. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-97088-2_7. URL https://doi.org/10.1007/978-3-319-97088-2_7
13. Jackson, D.: Promoting phenotypic diversity in genetic programming. In: PPSN 2010 11th International Conference on Parallel Problem Solving From Nature. Lecture Notes in Computer Science, vol. 6239, pp. 472–481. Springer, Krakow, Poland (2010). https://doi.org/10.1007/978-3-642-15871-1_48
14. Pantridge, E., Helmuth, T.: Solving novel program synthesis problems with genetic programming using parametric polymorphism. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'23, pp. 1175–1183. Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3583131.3590502>
15. Pantridge, E., Helmuth, T., Spector, L.: Functional code building genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'22, p. 1000–1008. Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3512290.3528866>, <https://doi.org/10.1145/3512290.3528866>
16. Pantridge, E., Spector, L.: Code building genetic programming. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference, GECCO'20, pp. 994–1002. Association for Computing Machinery, New York, NY, USA (2020). <https://doi.org/10.1145/3377930.3390239>
17. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. *Trans. Am. Math. Soc.* **74**(2), 358–366 (1953). <http://www.jstor.org/stable/1990888>
18. Spector, L., Klein, J., Keijzer, M.: The push3 execution stack and the evolution of control. In: GECCO 2005: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, vol. 2, pp. 1689–1696. ACM Press, Washington DC, USA (2005). <https://doi.org/10.1145/1068009.1068292>, <http://www.cs.bham.ac.uk/~wbl/biblio/gecco2005/docs/p1689.pdf>
19. Turner, A.J., Miller, J.F.: Neutral genetic drift: an investigation using cartesian genetic programming. *Genet. Program. Evol. Mach.* **16**, 531–558 (2015)

Chapter 8

Sharpness-Aware Minimization in Genetic Programming



Illya Bakurov, Nathan Haut, and Wolfgang Banzhaf

Abstract Sharpness-Aware Minimization (SAM) was recently introduced as a regularization procedure for training deep neural networks. It simultaneously minimizes the fitness (or loss) function and fitness sharpness. The latter serves as a measure of the nonlinear behavior of a solution guiding toward solutions that lie in neighborhoods with uniformly similar loss values across all fitness cases. In this contribution we adapt SAM for tree Genetic Programming (TGP) by exploring the semantic neighborhoods of solutions using two simple approaches. By perturbing input and output of program trees, sharpness can be estimated and used as a second optimization criterion during the evolution. To better understand the impact of this variant of SAM on TGP, we collect numerous indicators of the evolutionary process, including generalization ability, complexity, diversity, and a recently proposed genotype-phenotype mapping to study the amount of redundancy in trees. The experimental results demonstrate that using any of the two proposed SAM adaptations in TGP allows (i) a significant reduction of tree sizes in the population and (ii) a decrease in redundancy of the trees. When assessed on real-world benchmarks, the generalization ability of solutions does not deteriorate.

8.1 Introduction

The automatic discovery of mathematical expressions to describe phenomena captured in data is an extremely valuable tool for accelerating scientific discovery since the mathematical expressions can be used to make predictions about the systems that

I. Bakurov (✉) · W. Banzhaf

Department of Computer Science and Engineering, Michigan State University, East Lansing, MI, USA

e-mail: bakurov1@msu.edu

W. Banzhaf

e-mail: banzhaf@msu.edu

N. Haut

Department of Computational Mathematics, Science, and Engineering, Michigan State University, East Lansing, MI, USA

e-mail: hautnath@msu.edu

generated the data and the expressions can be directly studied to extract new insights into the system. There are many approaches for finding equations that fit data: linear regression, polynomial regression, SINDy [7], neural-symbolic regression [6], symbolic regression [19], etc. Genetic programming (GP) is a popular method for finding equations that fit data since it allows great flexibility for the discovery of nonlinear behaviors in data while also being effective in small data scenarios, unlike deep learning (DL) approaches which generally require large training datasets. This ability of GP to be effective in small data scenarios is likely related to an evolutionary algorithm's bias for simple solutions, as simple solutions are less likely to overfit [5]. Even so, in small data scenarios, the models are naturally underconstrained in the interstitial spaces between the training data points, which means that surprising and unexpected behavior can occur when interpolating. Ideally, we would want the models to be at least stable (smooth) when interpolating, otherwise the generalization ability of a model can be severely reduced.

Some GP methods have been proposed to help stabilize model predictions to improve the robustness against overfitting in small data scenarios, such as order of nonlinearity [38], model curvature [35], random sampling technique (RST) [13], RelaxGP [8], and overfit repulsors [36]. Order of nonlinearity and model curvature are approaches that attempt to consider model properties to predict overfitting [35, 38]. Random sampling attempts to reduce the risk of overfitting by ensuring that no model sees the whole dataset in a single generation [13]. RelaxGP minimizes the risk of overfitting by assigning no additional fitness for getting closer than some threshold to the response data [8]. The use of overfit repulsors stores known overfit models and promotes the evolution of individuals that are different in semantic space [36].

Here we propose to use model properties to predict and discourage overfitting while using synthetic data to catch models that are unstable in the interstitial spaces around the known training data. This approach is inspired by the Sharpness-Aware Minimization (SAM) technique recently developed in DL for neural networks [10] and also applied in a different way in [40] for feature construction. In [10], the authors demonstrate that SAM can promote generalization in neural networks. This is achieved by searching for neural network model parameters that lie in neighborhoods of low sharpness.

We introduce two new methods for selecting against sharpness in genetic programming: input-based SAM (SAM-In) and output-based SAM (SAM-Out). In this work, we aim to:

- Adapt sharpness-aware minimization from a deep learning regularization method to be compatible with genetic programming in an efficient way;
- Reduce the risk of overfitting in GP by rewarding the smoothness of models rather than just the accuracy;
- Improve the stability of models by penalizing models that vary significantly in their response surfaces in neighborhoods in the fitness landscape.

8.2 Related Work

8.2.1 Sharpness-Aware Minimization in Deep Learning

DL models commonly used in computer vision are often comprised of hundreds of megabytes and millions of parameters, whereas large language models are made of billions of parameters and their size is measured in gigabytes. Training such complex systems is a big challenge for deep learning, but the high demand for computational resources and data availability are not the only constraints. DL models are particularly prone to memorizing the training data rather than generalizing the learned patterns to unseen data (i.e., overfitting). Several techniques are commonly used to mitigate overfitting in DL and foster convergence: penalizing loss/fitness functions for disproportional high weights (such as L1 and L2 penalties), setting the weights connecting randomly selected neurons in some layer(s) to zero during a given training iteration, usage of varied data augmentation techniques to expose models to more variation, normalization of data batches that pass through the network, etc. Some DL researchers focus their attention on how a variety of neural architecture choices (such as depth, width, network architecture, optimizer selection, connectivity patterns, and batch size) affect the geometry of fitness landscapes and relate this to network generalization. For instance, Li et al. concluded [21] that landscape geometry significantly affects the generalization ability of a system and that larger network depth, batch size, and usage of sequential connections without shortcut connections produce sharper fitness landscapes and, consequently, negatively impact the generalization ability of networks.

Motivated by the connection between the fitness landscape and generalization, P. Foret et al. proposed a novel procedure that improves model generalization by simultaneously minimizing the loss value and sharpness of the loss landscape by promoting parameters that lie in neighborhoods having uniformly low loss value [10]. Specifically, the authors add a regularization term to the loss function that is a measure of how training loss can be increased by *moving* weights W to a *nearby* parameter value $W + \varepsilon$, where ε represents the perturbation. The authors demonstrate that SAM improves model generalization ability across a range of widely studied computer vision tasks and provides robustness to label noise on par with that provided by SOTA procedures that specifically target learning with noisy labels. This procedure was called Sharpness-Aware Minimization (SAM) and constitutes the main inspiration for the approach proposed in this manuscript.

8.2.2 Semantic Awareness in Genetic Programming

Traditional crossover and mutation operators used in GP rely on structural (i.e., genotypic) transformations of parent individuals, without knowledge of their effects

on the offspring's behavior. But a mere replacement of one node can lead to arbitrarily large changes in the behavior of a program (e.g., replacing—with $*$ in $x_1 + x_2 - x_3$).

Since 2012 GP researchers have incorporated semantic awareness into GP.¹ Moraglio et al. [24] proposed geometric semantic operators (GSOs) that allow to transform genotypes of parents in such a way that the effect on the semantics of the offspring is known, unlike what happens with standard GP operators [18]. A remarkable characteristic of GSOs is that the error surface on the training data for any supervised machine learning problem is unimodal and therefore easy to search. This holds independently of the size and complexity of the dataset [34]. GSOs allowed GP to achieve better accuracy in numerous real-world applications, even when compared with other ML approaches [2, 20, 37].

Vanneschi et al. [37] demonstrated that using a sigmoid-bounding function in geometric semantic mutation (GSM) helps to stabilize the learning process and increase the generalization ability of solutions. In this sense, the semantics of the offspring resulting from a GSM will surround the semantics of the parent within a user-controlled range $[-ms, ms]$, which can be interpreted as a step in any direction of the semantic space in a box of side ms . Recently, Bakurov et al. [3] proposed a normalization procedure for GSM that reduces the size of the programs and overcomes some saturation issues associated with the sigmoid function adopted in [37], which resulted in more accurate and simple solutions. This is the GSM variant used in our study. Gonçalves et al. introduced the concept of the semantic neighborhood—the set of neighbors reachable from a given solution when a GSM is applied to it—and proposed a stopping criteria based on neighborhood properties [12]. Competitive generalization was achieved within significantly fewer generations, and consequently in smaller solutions. This suggests that information collected from the semantic neighborhood can bring notable value for the evolutionary process.

8.2.3 Noisy Data and Fitness Functions

A common problem in machine learning is overfitting, where the ML method focuses too much on chasing perfect accuracy and the model only learns to memorize the training data but fails to identify generalizable patterns. One method to avoid overfitting is to utilize a noisy fitness function, which is a fitness function that has noise introduced so that it is not exact and thus prevents the machine learning algorithm from chasing a stationary zero error solution in the search space. In evolutionary computation it has been shown that noisy fitness evaluations can be beneficial in rugged landscapes by allowing the search to more easily escape local optima and navigate toward a global optimum [31]. Also, evolutionary computation methods have been

¹ The term *semantics* defines the vector of output values of a candidate solution (program), calculated on the training observations [24]. Following this notion, a candidate solution in GP is a point in a multidimensional *semantic space*, where the dimensionality is equal to the number of observations in the training set.

found to be robust in “uncertain environments” where noise is naturally present in the environment [16, 29]. While somewhat different from a noisy fitness function where noise is intentionally injected into the fitness function, it demonstrates that noise is not necessarily a problem for evolutionary computation and may sometimes even be a benefit.

In GP, a similar approach has been to find a “good-enough” fit such that the error is within some threshold rather than trying to find a perfect fit with zero error. One such approach is RelaxGP [8], where the authors change the fitness to instead consider an upper and lower bound for each response value rather than trying to perfectly fit each response value. In GP, random sampling has also been shown to have similar benefits to noisy fitness evaluations since it improves the generalization of model and discourages memorization of the whole training set [13, 26].

8.2.4 SAM in GP

Very recently, sharpness was also incorporated in GP [40] within the scope of feature construction. In that work, sharpness was measured by introducing Gaussian noise to each node at all layers of the feature trees. The sharpness of an individual was then determined using the difference in loss after the noise was introduced. The results demonstrate that sharpness can indeed be used to reduce the size of GP trees and mitigate their tendency to overfit. However, one potential drawback of this approach is that it is computationally expensive since it computes sharpness using noise at each layer of the trees.

8.3 The Proposed Approach

Here we evaluate two approaches for estimating the sharpness of a given tree in GP. Because of the fundamental difference between artificial neural networks (ANN) and TGP, a direct transfer of SAM from the former to the latter is impracticable. Typically, ANNs have a predefined (fixed) architecture (e.g., number of layers and nodes per layer) and the search for an optimal solution consists of finding an optimal set of weights (a collection of real values) that minimizes the loss; therefore, for ANNs, the loss landscape is continuous.

Although there exist some successful attempts to include learnable weights in TGP [27, 28, 32], TGP typically is comprised of discrete structures (program elements), commonly divided into two groups: (i) terminals, which represent the input features of the problem and numeric constant values, and (ii) functions, which represent the operations to be performed on terminals. The search for an optimal solution is thus conducted in a discrete space made up of combinations of these tree data structures, that can grow or shrink dynamically in size. Given these differences, we use two approaches to adapt SAM for TGP. The first acts on the input of a tree by

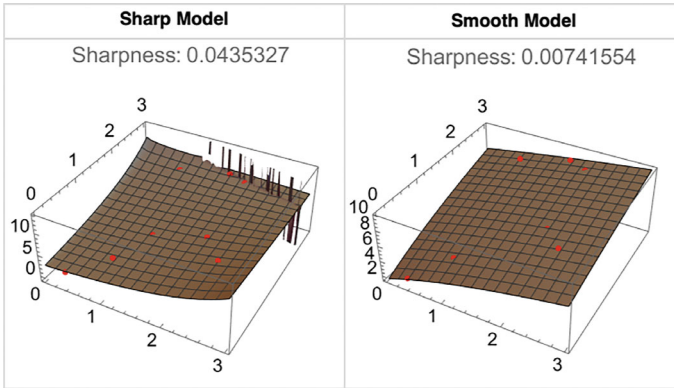


Fig. 8.1 A sharp and a smooth model are compared along with the values that were returned by the SAM-IN metric, showing that the sharper model is clearly identified by the metric

randomly perturbing both constants and input features of the model; in other words, it adds noise to a tree’s terminal nodes. The second approach consists of randomly perturbing the output of a tree. The subsections below explain each approach in more detail.

The goal of these methods is to identify and penalize those models of the population that exhibit strongly nonlinear behavior in the fitness landscape (sharpness). An example of the behavioral difference between two models is shown in Fig. 8.1, where we depict two models with similar accuracy on the training data. However, one is not stable in some regions of the fitness landscape, while the other is very smooth and stable. Training points are shown as red dots on the model response surfaces and sharpness, calculated by our metric, is listed for each model in the labels above the plots. The values show that the sharpness metric would be able to identify this unstable model and remove it from a population. Unknowingly selecting and using a sharper model would be detrimental since it has very unstable behavior in certain regions of input space.

8.3.1 SAM on Input (SAM-IN)

Our first approach, called SAM-IN, quantifies the sensitivity of each tree in the population with respect to random perturbations in the terminals, both in variables and constants. This sensitivity is calculated by measuring the absolute fitness differential after applying random noise of magnitude ε to a subset of input training data instances n and all of the model constants. The noise introduced to the variables is a function of the standard deviation of the data in that variable. Then, the trees are executed a second time in order to obtain the fitness on the perturbed training cases. This quantity is then used as a second selection criterion during an evolutionary run. Selection is

performed by using randomized double tournament selection, where the order of objectives is randomized for every selection. We expect that high sensitivity toward input noise indicates that a given tree is likely overfit and thus should have a lower selection preference when compared to a tree with a smaller sensitivity. Algorithm 8.1 is an implementation of SAM-IN. Mathematically, the SAM-IN metric is represented by Eq. (8.1), where n is the number of perturbations (in this work we just use just one perturbation), x is the input data, y is the response data, ε is the noise introduced to the input data, F is the model, $\tilde{F}_i$ is the model with perturbed constants, and Corr is a function for computing Pearson's correlation coefficient depicted in Eq. (8.2). In Eq. (8.2), y is the target response, $\hat{y}$ is the model response, and N is the number of data points.

$$S_{in} = \frac{\sum_{i=1}^n |\text{Corr}(F(x), y)^2 - \text{Corr}(\tilde{F}_i(x + \varepsilon_i), y)^2|}{n} \quad (8.1)$$

$$\text{Corr}(\hat{y}, y) = \frac{\sum_{i=1}^N (y_i - \bar{y})(\hat{y}_i - \bar{\hat{y}})}{\sqrt{\sum_{i=1}^N (y_i - \bar{y})^2 \times \sum_{i=1}^N (\hat{y}_i - \bar{\hat{y}})^2}} \quad (8.2)$$

Algorithm 8.1 SAM on Input *Revision in progress*

Require: perturbations number $n > 0$, perturbation magnitude $\varepsilon > 0$, training dataset D

```

1:  $\vec{\sigma} \leftarrow$  Compute the standard deviation of each feature in  $X$ 
2: for each generation in the evolutionary run do
3:    $X_s \leftarrow$  Select a random sample of size  $n$  from  $X$ 
4:    $X_{s+\varepsilon} \leftarrow$  Make a copy of  $X_s$ 
5:   for  $i = 1$  to  $\text{LENGTH}(X_{s+\varepsilon})$  do
6:      $\varepsilon_i \leftarrow$  generate a random value in  $[-\varepsilon \vec{\sigma}, \varepsilon \vec{\sigma}]$ 
7:      $X_{s+\varepsilon}[i] = X_{s+\varepsilon}[i] + \varepsilon_i$ 
8:   end for
9:   for each tree  $t$  in the population do
10:     $t_\varepsilon \leftarrow$  make a copy of  $t$ 
11:    for each constant  $c$  in  $t_\varepsilon$  do
12:       $\varepsilon_c \leftarrow$  generate a random value in  $[-\varepsilon, \varepsilon]$ 
13:       $c = c + \varepsilon_c$ 
14:    end for
15:     $f^t \leftarrow$  compute the fitness of  $t$  on  $X_s$ 
16:     $f^{t+\varepsilon} \leftarrow$  compute the fitness of  $t_\varepsilon$  on  $X_{s+\varepsilon}$ 
17:     $S_{SAM-IN}^t \leftarrow$  compute sharpness of  $t$  as  $|f^t - f^{t+\varepsilon}|$ 
18:  end for
19: end for

```

Unlike the original SAM in DL, or the SAM variant introduced in [40], SAM-IN introduces noise in the terminals rather than all of the connections (red arrows in Fig. 8.2). This is computationally less expensive than the SAM-GP introduced in [40] and better reflects the idea behind generalization ability: the ability to provide accurate and stable predictions for new, previously unseen data, drawn from the same

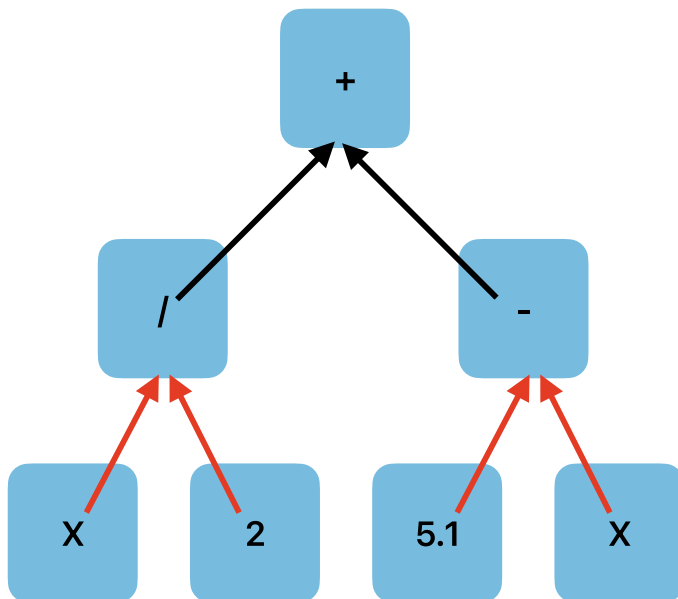


Fig. 8.2 An example of a GP tree. The red arrows indicate the location of injected noise

distribution as the one used for training. The small noise will propagate through all layers of a tree and helps to measure the potential for accurate and stable predictions.

8.3.2 SAM on Output (SAM-OUT)

Our second approach, called SAM-OUT, builds upon the mechanics of the GSM operator and the notion of semantic neighborhood introduced by [11] and described in Sect. 8.2.2. Given a reference individual (tree), it generates n semantic neighbors using GSM with $ms = \varepsilon$, where n and ε are hyper-parameters of SAM-OUT. The fitness of each semantic neighbor is calculated and the sharpness of the reference individual is estimated as the variance in the semantic neighborhood. A large variance/instability in the semantic neighborhood indicates that a given reference individual is likely to overfit and, thus, should have a lower selection preference when compared to a tree with a smaller variance. Algorithm 8.2 contains an implementation of SAM-OUT. We use the normalized-GSM as proposed in [3] given its notable stability and simplicity. Mathematically, SAM-OUT is represented in Eq. (8.3), where x represents input data, y the response data, ε is the noise introduced to the model output, F is the model, and Corr is a function for computing Pearson's correlation coefficient shown in Eq. (8.2).

$$S_{out} = \frac{\sum_{i=1}^n |Corr(F(x), y)^2 - Corr(F(x) + \varepsilon_i, y)^2|}{n} \quad (8.3)$$

Note that we do not need to construct and execute a random tree to sample from the semantic neighborhood of a reference individual, as the semantic neighbors are only used for sharpness assessment. Because semantic neighbors will not be used anywhere else in the process, SAM-OUT (unlike SAM-IN) can fully operate in semantic space, which renders it significantly more efficient. Thus one can generate a semantic neighbor by summing the semantics of the reference individual with a randomly generated vector of values centered at zero and bounded in $[-\varepsilon, \varepsilon]$. It is important to note that there is no need for additional execution of the reference individual to extract its semantics, as it is already done when performing fitness evaluation.

Algorithm 8.2 SAM on Output

Require: perturbations number $n > 0$, perturbation magnitude $\epsilon > 0$, training dataset X

```

1: for each generation in the evolutionary run do
2:   for each tree  $t$  in the population do
3:      $\hat{y}_t \leftarrow$  Compute the output of the tree  $t$  on  $X$ 
4:      $\sigma_{\hat{y}_t} \leftarrow$  Compute the standard deviation of  $\hat{y}_t$ 
5:      $\vec{f}_n^t \leftarrow$  empty vector for storing fitness of  $n$  semantic neighbors of  $t$ 
6:     for each perturbation  $i$  in  $n$  do
7:        $\hat{y}_{t+\epsilon}^i \leftarrow$  Make a copy of  $\hat{y}_t$ 
8:       for  $j = 1$  to  $\text{LENGTH}(\hat{y}_{t+\epsilon}^i)$  do
9:          $\epsilon_{(j,i)}^t \leftarrow$  generate a random vector in  $[-\epsilon\sigma_{\hat{y}_t}, \epsilon\sigma_{\hat{y}_t}]$ 
10:         $\hat{y}_{t+\epsilon}^i[j] = \hat{y}_{t+\epsilon}^i[j] + \epsilon_{(j,i)}^t$ 
11:      end for
12:       $\vec{f}_n^t[i] \leftarrow$  compute the fitness of the  $i^{th}$  semantic neighbor  $\hat{y}_{t+\epsilon}^i$ 
13:    end for
14:  end for
15:   $S_{SAM-OUT}^t \leftarrow$  compute sharpness of  $t$  as  $\sigma^2(\vec{f}_n^t)$ 
16: end for
```

A subtlety arises when using R^2 as fitness function as opposed to RMSE fitness. In this case SAM-OUT is not functionally identical to introducing noise to the target vector, as would occur when using noisy fitness functions. This is shown in Table 8.1 where the same error with RMSE is observed when adding the noisy vector, ε , to the predicted target, $\hat{y}$, as when subtracting ε from the target y . This is, however, not the case when employing R^2 as fitness function, which means that adding noise to the output is different here from adding noise to the target.

Table 8.1 Comparison of SAM-OUT versus Noisy target for RMSE and R^2 . y represents the target variable, $\hat{y}$ represents the model predicted target, and ε represents a normally distributed noisy vector. ε was randomly initialized and then used in the same state for each evaluation in the table

Noise location	RMSE value	R^2 value
$y + \varepsilon$	2.931	0.431
$y - \varepsilon$	2.831	0.294
$\hat{y} + \varepsilon$	2.831	0.817
No noise	1.957	0.847

Table 8.2 Details about the four real-world datasets used

Dataset	#Instances	#Features
Boston [14]	506	13
Heating [33]	768	8
Diabetes [9]	442	10
Concrete [39]	1005	8

8.4 Experimental Settings

We assess our method on data from four real-world regression problems and four popular synthetic functions. The real-world problems are described in Table 8.2. As for the latter, given a synthetic function $f(X)$, we randomly sample 100 data points in a 2-dimensional grid (x), under uniform distribution, where each dimension corresponds to an input feature (x_1 and x_2). The third dimension corresponds to the target of the prediction y and is obtained as $f(x) = y$. The synthetic problems are described in Table 8.3. In each run, we use different train-test partitions of the data. For real-world problems, we randomly sample 70% observations for training, while the remaining 30% are held out for testing. For synthetically generated problems, we use 50% for training and 50% for testing.

Table 8.4 lists the hyper-parameters (HPs) used in this study, along with cross-validation settings. The HPs were selected following common practices found across the literature to avoid a computationally demanding tuning phase. R^2 was used as fitness function [15, 17, 22] as it was found to converge faster, generalize better, even when only a few data points are available. Programs allowing invalid operations (like dividing by 0 or taking a square of a negative value) were automatically assigned a low fitness value. Although operator protection ensures a valid numerical output for any input, it can also produce unexpectedly high/low (i.e., sharp) predictions as shown in [17]. This would inevitably bias our sharpness-aware approach to avoid programs containing such operators. Thus, to compare sharpness-aware minimization and standard GP on a fair basis, none of the operators is protected in this study. Two rounds of tournament selection are used to select individuals, with sizes 6 and

Table 8.3 Details about the four synthetic datasets used. In our experiments, $D = 2$

Name	Function	$x_i \in$
Levy	$f(\mathbf{x}) = \sin^2(\pi w_1) + \sum_{i=1}^{D-1} (w_i - 1)^2 (1 + 10 \sin^2(\pi w_i + 1)) + (w_d - 1)^2 (1 + \sin^2(2\pi w_d)),$ $w_i = 1 + \frac{x_i - 1}{4}$	$[-10, 10]$
Ackley	$f(\mathbf{x}) = -a \exp\left(-b \sqrt{\frac{1}{D} \sum_{i=1}^D x_i^2}\right) - \exp\left(\frac{1}{D} \sum_{i=1}^D \cos(c x_i)\right) + a + \exp(1),$ $a = 20, b = 0.2, c = 2\pi$	$[-32.768, 32.768]$
Rastrigin	$f(\mathbf{x}) = 10D + \sum_{i=1}^D (x_i^2 - 10 \cos(2\pi x_i))$	$[-5.12, 5.12]$
Rosenbrock	$f(\mathbf{x}) = \sum_{i=1}^{D-1} (100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$	$[-2.048, 2.048]$

Table 8.4 Summary of the hyper-parameters. Note that $P(C)$ and $P(M)$ indicate the crossover and the mutation probabilities, respectively

Parameters	Values
N ^o Train/test split	Real-world problems: 70/30%; synthetic problems: 50–50%
Cross-validation	Monte-Carlo (repeated random subsampling)
N ^o runs	60
N ^o generations	50
Population's size	100
Functions (F)	$\{+, -, \times, / \sin(x), \cos(x), \tanh(x), x^2, x^{-1}, \sqrt{x}, e^x, \log(x)\}$
Initialization	Ramped Half&Half (RHH) with max depth of 5
Selection	double tournament with sizes 6 and 3, respectively
Genetic operators	{swap crossover, subtree mutation}
$P(C)$	0.8
$P(M)$	0.2
Maximum depth limit	Not applied
Stopping criteria	Maximum N ^o generations
SAM noise magnitude (ε)	{0.1, 0.2, 0.5, 1.0}
SAM number of noisy perturbations (n)	{10, 20, 50}

3, respectively. At each selection event, a criterion for selection is chosen at random (either training fitness or sharpness). The returned individual is the one that exhibits the highest R^2_{train} and the smallest sharpness.

Recently, Banzhaf and Bakurov [4] proposed an effective approach to map a genotype to its phenotype in TGP by removing semantically ineffective code from the former. This genotype-phenotype mapping (GPM) is highly efficient as it is built upon the mechanics of the fitness evaluation (which inevitably needs to take place), avoiding, therefore, redundant function calls and calculations. The authors show that smaller phenotypes are hidden within larger genotypic trees, easily extracted to facilitate interpretability. They also studied the population dynamics of both genotypes and phenotypes and concluded that population behavior is normally based on a scarce number of unique and small phenotypes, which—curiously—happens even when evolution is more explorative. The authors observed that the growth rate of phenotype size is notably smaller than that of genotypes. All of this suggests that using the GPM approach presented in [4], one can extract valuable insights about evolutionary dynamics. To make the best use of these results and allow an unbiased assessment of the impact of SAM in terms of bloat, no limit to tree depth was applied during evolution. Initially, we did not have any assumption about SAM’s impact on solution size; instead, we wanted to allow for more degrees of freedom for SAM to manifest its features.

To conduct our experiments, we use the General Purpose Optimization Library [1]. GPOL is a flexible and efficient multi-purpose optimization library in Python that covers a wide range of stochastic iterative search algorithms, including GP. Its modular implementation allows for solving optimization problems, like the one in this study, and easily incorporates new methods. The library is open-source and can be found by following [this link](#). The implementation of the proposed approach can be found there.

8.5 Experimental Results

Table 8.5 reports the ranks on the generalization ability obtained by different SAM variants and baseline TGP. To build the table, the R^2 fitness values on the test data observed in the last generation were ranked in ascending order. Then, the ranks were averaged by algorithm type (standard GP and SAM variants). From the table, one can observe that several of the proposed SAM variants tend to rank the best more often than standard GP in terms of generalization ability. When comparing two distinct SAM approaches, SAM-IN ranks above its SAM-OUT counterpart. Additionally, from this table, we can extract the reference parameters for the two SAM variants. Both tend to produce better ranks using mild perturbations (0.1) and small neighborhood (10 and 20 for SAM-IN and SAM-OUT, respectively). From the table, these are $\text{SAM-IN}_{10}^{0.1}$ and $\text{SAM-OUT}_{20}^{0.1}$. These SAM configurations were used in all figures of this section as the default SAM-IN and SAM-OUT, depicted as dashed dark blue and dark red lines.

Table 8.5 Generalization ability ranks achieved by each version of SAM and standard GP. Large values correspond to high fitness ranks. Colors reflect fitness: the green end of the green-yellow-red color scale highlights the highest (best) ranks observed in an experiment; the red end of the scale highlights the smallest (worse) ranks. The last row averages results of each column (SAM experiment)

Algorithm		$SAM_{10}^{0.10}$		$SAM_{20}^{0.10}$	$SAM_{50}^{0.10}$	$SAM_{10}^{0.20}$	$SAM_{20}^{0.20}$	$SAM_{50}^{0.20}$	$SAM_{10}^{0.50}$	$SAM_{20}^{0.50}$	$SAM_{50}^{0.50}$	$SAM_{10}^{1.0}$	$SAM_{20}^{1.0}$	$SAM_{50}^{1.0}$	$SAM_{10}^{0.50}$	$SAM_{20}^{0.50}$	$SAM_{50}^{0.50}$	$SAM_{10}^{1.0}$	$SAM_{20}^{1.0}$	$SAM_{50}^{1.0}$
SAM GP	IN	7.38	9.88	6.5	8.88	8.63	3.75	8	7	6.13	5.38	6.13	7.25	6.13	7.25	6.13	6.13	7.25	6.13	6.13
	OUT		4.63	8.5	6.13	8.13	4.75	6.5	6.5	7	6.88	7.13	6.63	5.25	6.63	5.25	5.25	6.63	5.25	5.25
	AVG		7.25	7.50	7.50	8.38	4.25	7.25	6.75	6.56	6.13	6.63	6.94	5.69	6.94	5.69	5.69	6.94	5.69	5.69

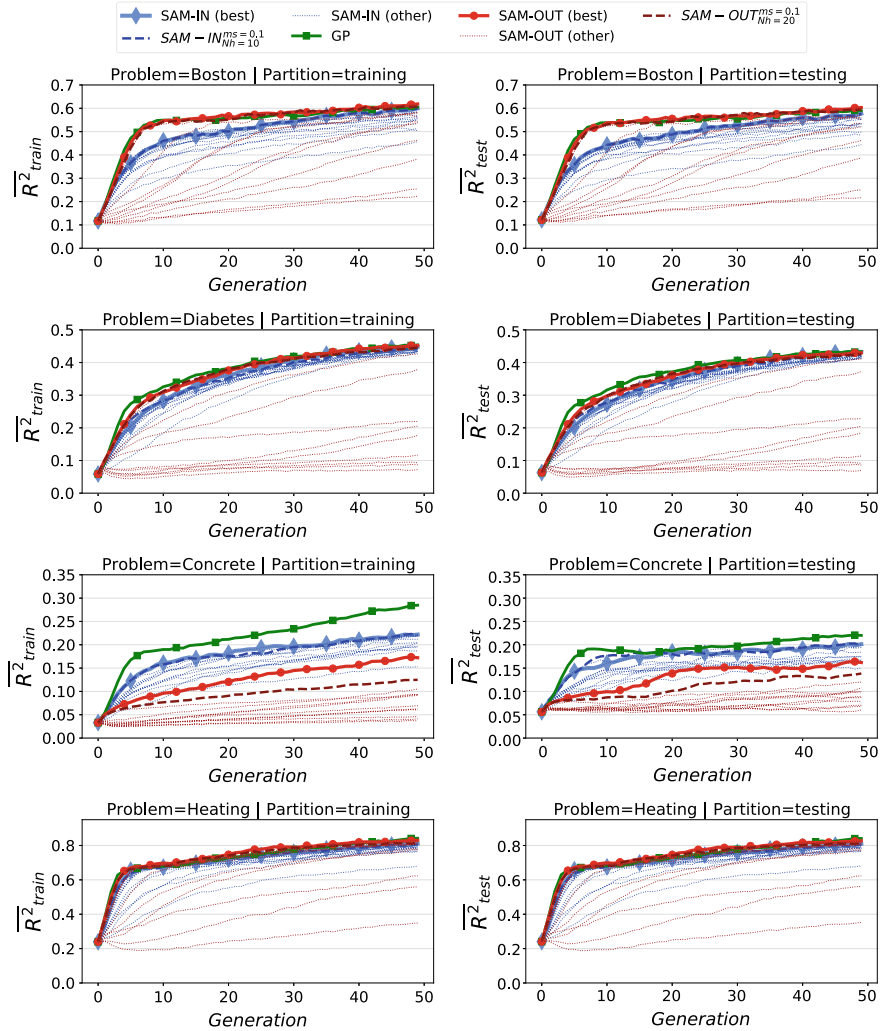


Fig. 8.3 Average population training (left) and test (right) fitness of SAM approaches versus standard GP on the 4 real-world datasets

Figures 8.3 and 8.4 show the average fitness in the population for real-world and synthetic problems, respectively. Each problem is depicted in a given row, whereas each column represents a given partition: training on the left, and test on the right. The solid green lines represent standard GP (in green). The solid red and royal blue lines represent the SAM-IN and SAM-OUT methods that achieved the best generalization ability on a given problem. The SAM-IN and SAM-OUT configurations that were found to score the best more often across all the problems are depicted as dashed blue and dark red lines (these are the recommended SAM configurations and were

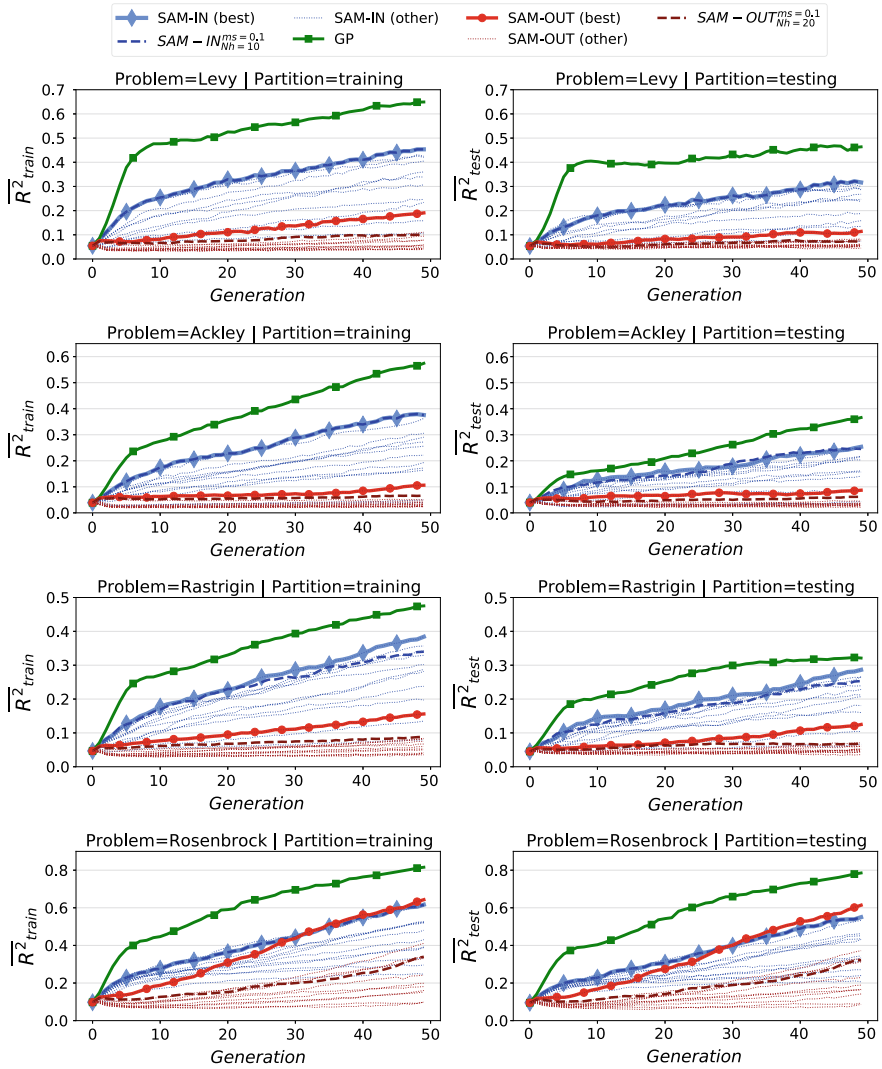


Fig. 8.4 Average population training (left) and test (right) fitness of SAM approaches versus standard GP on the 4 synthetic datasets

obtained from Table 8.5). The remaining SAM-IN and SAM-OUT configurations are depicted as thin dotted lines, in blue and dark red, respectively. When looking at Fig. 8.3, the average population fitness is roughly the same for standard GP and the SAM approaches on Boston, Diabetes, and Heating. On the Boston problem, however, SAM-IN shows a slower convergence. On the Concrete problem, the highest fitness values are observed for standard GP, followed by SAM-IN and ultimately SAM-OUT. This particular problem seems more challenging to generalize on test

data for all the GP variants. It is also relevant to point out that the recommended SAM configurations tend to overlap with the best SAM configuration for particular problems, highlighting their robustness across problems.

When looking at Fig. 8.4, we observe a higher average fitness in the population when using standard GP on all the problems, across both partitions. SAM-IN follows standard GP and SAM-OUT achieves the worst average fitness values on both partitions. This is not the case, however, on the Rosenbrock problem where the best SAM-IN and SAM-OUT achieve competitive performance. In conclusion, the proposed SAM approaches tend to produce similar population training and test fitness on real-world problems when compared to standard GP, while on synthetic low dimensional problems they report worse fitness.

Figures 8.5 and 8.6 show the training and test fitness of the elite individuals, averaged across the evolutionary runs. The figure arrangement conforms to that of Figs. 8.3 and 8.4. When looking at Fig. 8.5, SAM-OUT converges faster and achieves better generalization when compared to both standard TGP and SAM-IN. Specifically, it achieves better test fitness than TGP on Boston and Heating problems, and similar fitness on Diabetes. Only on the Concrete problem, which seems to be challenging to generalize for all the algorithms, SAM-OUT achieves worse generalization. SAM-IN produces similar results to TGP across three problems (Boston, Diabetes, and Concrete) and scores the worst on Heating. For the elite fitness on synthetic problems, depicted in Fig. 8.6, standard TGP provided the best generalization, followed by SAM-IN which achieved the highest test scores on Rastrigin and similar scores on the Levy problem. SAM-OUT achieved the worst generalization ability, except on the Rosenbrock problem where it outperformed SAM-OUT but remained below TGP. In conclusion, the proposed SAM approaches tend to generate elites with equivalent or superior training and test fitness on real-world problems, while on synthetic low dimensional problems they report worse fitness.

We also recorded the average length of trees (node count) in the population across the evolutionary runs for all problems. Additionally, we measure the average difference between the whole genotype and the behavioral determinants (phenotype) of the individuals in the population using the GPM proposed in [4]. The latter was used to compute the ratio of redundant code (introns) in the trees. We display those results in Figs. 8.7 and 8.8 for the real-world and synthetic problems, respectively. The left column represents the average length, whereas the column on the right represents the average proportion of redundancy. Each problem is depicted in a given row. The results show that both SAM methods tend to produce notably smaller trees and, within those trees, the amount of nodes that do not contribute to the behavior is significantly smaller (i.e., there is a higher utilization of the code). Usually, SAM-OUT produces smaller trees with less redundancy than SAM-IN. Although this happens in real and synthetic problems, it is particularly notable for the latter.

Figure 8.9 reports the number of seconds per generation (first vertical axis) and the population size (second vertical axis) averaged across 10 runs on two problems: Levy and Concrete. The experiment was performed on an Intel(R) Core(TM) i7-8750H CPU processor with 2.20GHz, and 16GB of RAM. Solid lines represent the time, and dashed lines represent the size. One can notice that both SAM approaches

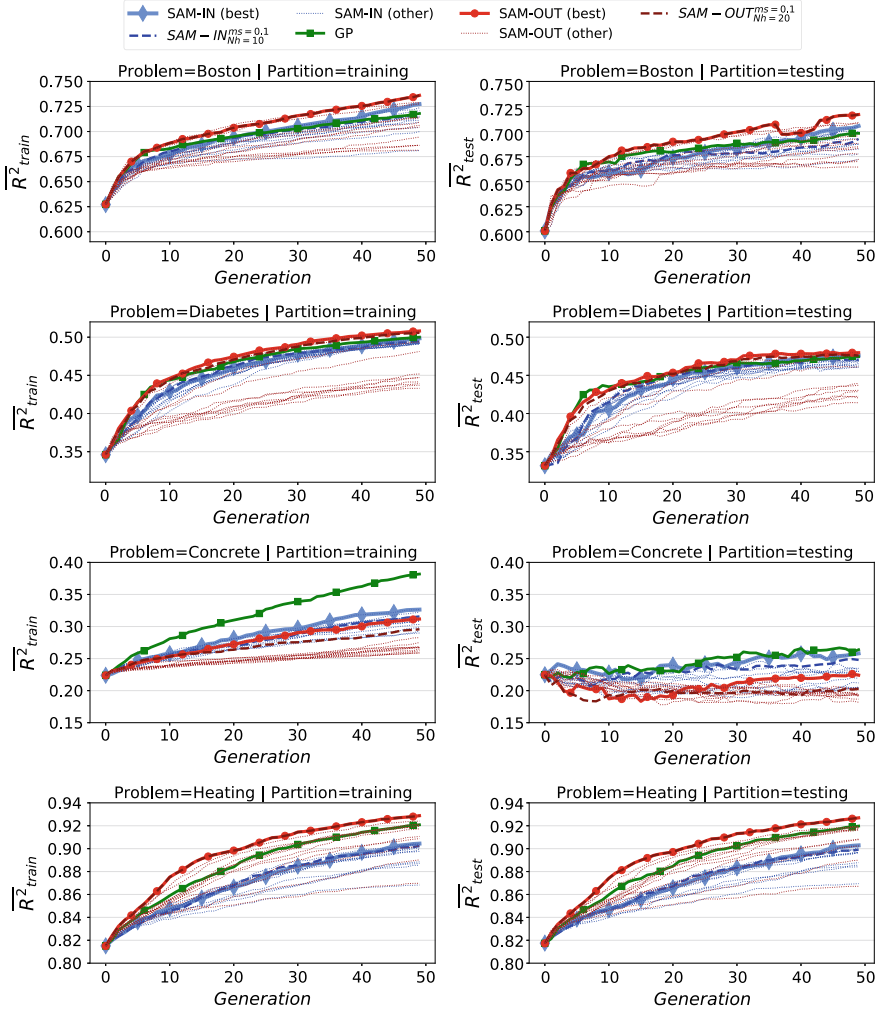


Fig. 8.5 Average best-performing model fitness of SAM approaches versus standard GP on the 4 real-world datasets

report smaller runtimes and average lengths of individuals than standard GP, with SAM-OUT notably more efficient. Smaller average length was already observed in Figs. 8.7 and 8.8. The efficiency of SAM-OUT is expected as it operates exclusively on the semantics of trees, without executing them; moreover, it provides notably smaller trees, which gives additional benefits in terms of runtime. SAM-IN is shown to outperform standard GP in terms of both runtime and average length.

We would like to point out a further potential benefit of optimizing against sharpness: In scenarios where the goal is to use the model to predict a global optimum, such as in an active design of experiment context. In those scenarios, developing a model

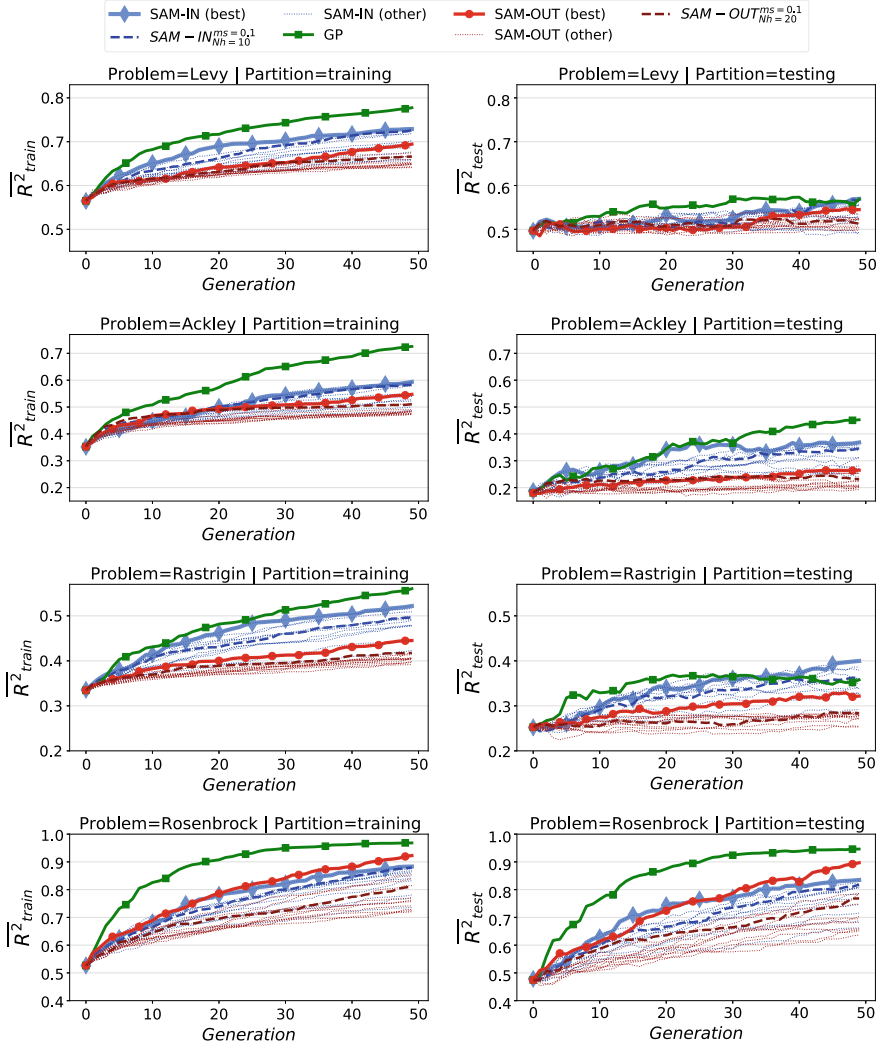


Fig. 8.6 Average best-performing model fitness of SAM approaches versus standard GP on the 4 synthetic datasets

with a smoother surface would make optimizing on the surface simpler and could lead to a convex surface which would make finding the global optima even trivial. This is demonstrated in Fig. 8.10, where a sparse training dataset of 50 points was gathered from the 2D Rastrigin function (left-most subplot), which is notoriously rugged, and then TGP along with the two SAM approaches were used to generate approximation models of the dataset. The resulting SAM models are remarkably smoother than that provided by TGP (second subplot from the left). Both of them are convex and exhibit

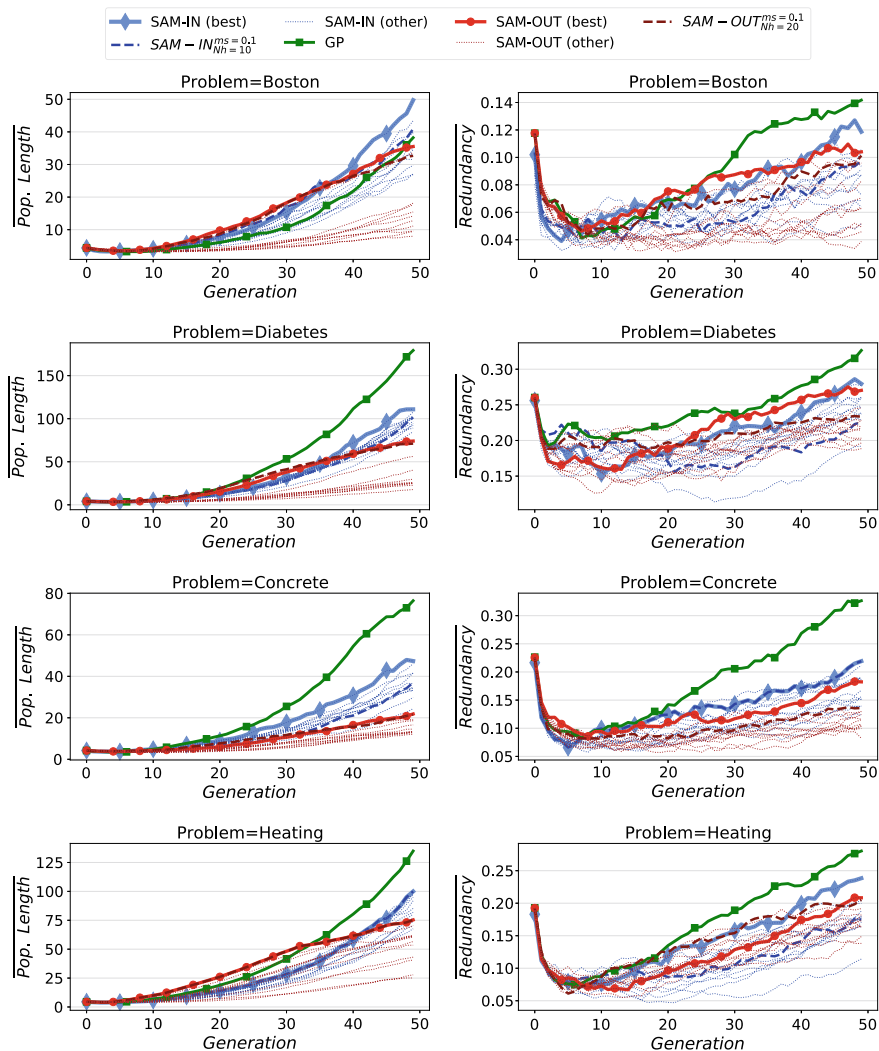


Fig. 8.7 Average length trees in the population (left) and the average amount of redundancy (right). The latter was calculated as the difference between the genotype and phenotype using the method proposed in [4]. The figure regards 4 real-world datasets

a global minimum in the same location as the target function. Notice, however, that SAM-IN generated a better approximation of the target surface, which is supported by the analysis of aggregated performances on Synthetic problems in Fig. 8.6.

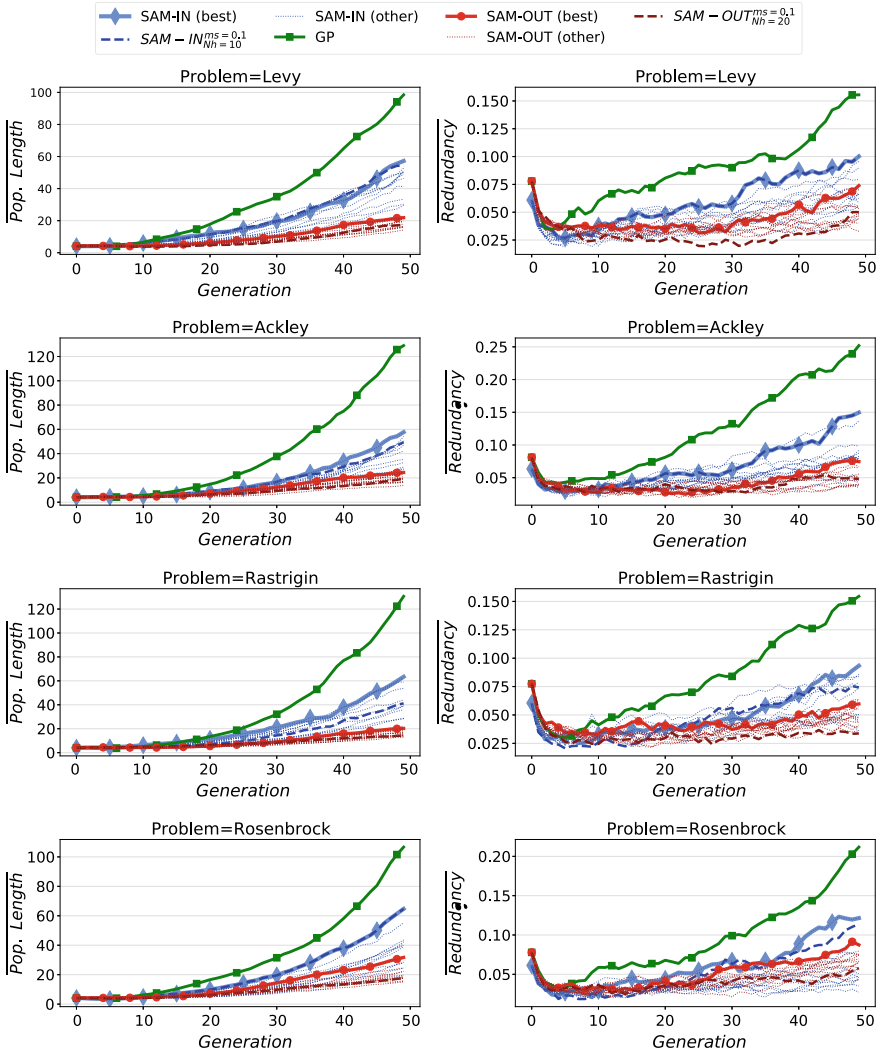


Fig. 8.8 Average length trees in the population (left) and the average amount of redundancy (right). The latter was calculated as the difference between the genotype and phenotype using the method proposed in [4]. The figure regards 4 synthetic datasets

8.6 Conclusions

In Machine Learning, the term *learning* often refers to the task of inducing a general pattern from a provided set of examples, instead of *memorizing* them. This ability is known as *generalization* and represents one of the main battlegrounds in the field.

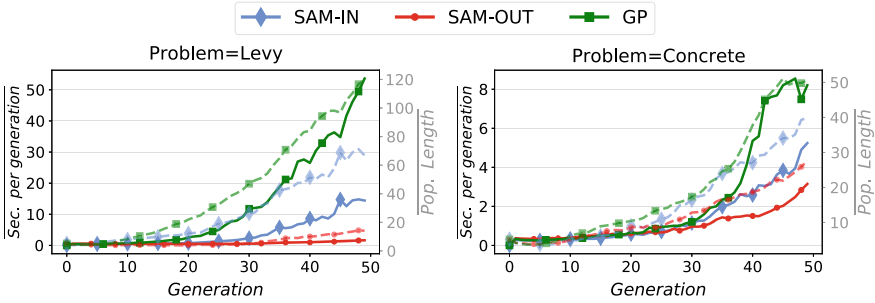


Fig. 8.9 Average number of seconds per generation (first vertical axis) and average population size (second vertical axis) on two problems: Levy and Concrete. The figure aggregates data from 10 runs executed on an Intel(R) Core(TM) i7-8750H CPU processor with 2.20GHz, and 16GB of RAM. Solid lines represent the time and dashed lines represent the size

In this study, we adapt the Sharpness-Aware Minimization (SAM), previously introduced in Deep Learning (DL) to improve model generalization by simultaneously minimizing the loss and sharpness of the loss landscape through promoting parameters that lie in neighborhoods of uniformly low loss values. To accommodate SAM in GP, we propose two methods. The first, called SAM-IN, measures sharpness by adding slight perturbations to the terminals of the programs (i.e., to both constants and input features). The second, called SAM-OUT, measures sharpness by applying slight perturbations to the output of the programs (i.e., the semantics). Both penalize individuals for large fitness changes upon perturbation. In the latter case, sharpness is assessed without the need to execute the program, which makes it computationally more efficient.

Comparing the two SAM approaches to standard GP, we observe that co-selecting individuals by fitness and SAM leads to evolution to find smaller and less redundant models. On top of that, it allows for faster training times. For real-world problems which are characterized by high data dimensionality and noisy data, this was achieved without degradation of generalization ability. These findings are complementary with those observed in [40], who analyzed tree size distribution and found their SAM approach produces smaller trees than the standard GP. Following their formulation, SAM can be seen as a regularization technique in GP; however, unlike many methods like Parsimony Pressure [23], Grand Complexity [25], or Dynamic Limits [30], SAM does not consider model size as an optimization target.

Subsampling led to faster evaluation of sharpness, but it may have caused the SAM metric to be unstable compared to evaluating sharpness using the whole dataset. The subsampled metric may overlook localized sharpness in models if the subsample happens to select points only in smoother regions. Future work could explore the trade-off between subsample size and stability in the sharpness metric.

Incorporating model sharpness as a metric to detect model overfitting and instability could be essential for improving the stability, generalization ability, and simplicity of models developed by GP, leading to more trustworthy models. If a model becomes

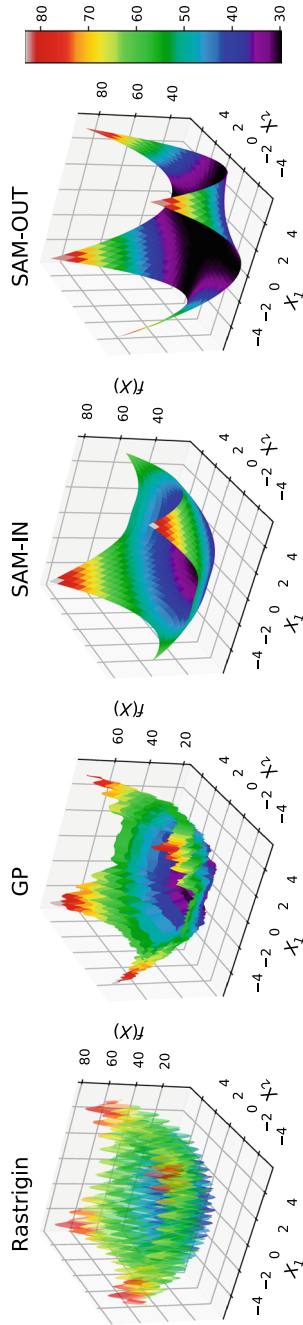


Fig. 8.10 An example of using TGP and sharpness to generate approximations of the original data generator Rastrigin function (left subplot). Penalizing for sharpness allows smoother approximations, which can make finding the global optima from those approximations trivial

unstable when being applied to make predictions and returns values that are wildly infeasible, a user would likely lose trust in the approach and seek out another method for developing models.

Our method for detecting sharpness is agnostic to the specific machine learning method being used since the perturbations only occur at the inputs and outputs of models. This means that our approach is general and can easily be adopted for other machine learning approaches as long as assumptions about training data distribution are not violated. We believe that the proposed sharpness metric should become part of the standard machine learning pipeline either during model training as a fitness metric or, minimally, as a post-processing selection criterion to ensure the models being deployed are stable.

Acknowledgements WB acknowledges support from the Koza Endowment fund administered by Michigan State University. Computer support by MSU's iCER high-performance computing center is gratefully acknowledged.

References

1. Bakurov, I., Buzzelli, M., Castelli, M., Vanneschi, L., Schettini, R.: General purpose optimization library (GPOL): a flexible and efficient multi-purpose optimization library in python. *Appl. Sci.* **11**(11), 4774 (2021). <https://www.mdpi.com/2076-3417/11/11/4774>
2. Bakurov, I., Castelli, M., Gau, O., Fontanella, F., Vanneschi, L.: Genetic programming for stacked generalization. *Swarm Evol. Comput.* **65**, 100913 (2021). <https://doi.org/10.1016/j.swevo.2021.100913>
3. Bakurov, I., Muñoz Contreras, J.M., Castelli, M., Rodrigues, N., Silva, S., Trujillo, L., Vanneschi, L.: Geometric semantic genetic programming with normalized and standardized random programs. *Genet. Program. Evol. Mach.* **25**(1) (2024). <https://doi.org/10.1007/s10710-024-09479-1>
4. Banzhaf, W., Bakurov, I.: On the nature of the phenotype in tree genetic programming. In: Li, X., Handle, J., et al. (eds.) *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2024)*, pp. 868–877. ACM Press, New York (2024). <https://dl.acm.org/doi/10.1145/3638529.3654129>
5. Banzhaf, W., Hu, T., Ochoa, G.: How the combinatorics of neutral spaces leads genetic programming to discover simple solutions. In: Winkler, S., Trujillo, L., Ofria, C., Hu, T. (eds.) *Genetic Programming Theory and Practice XX*, pp. 65–86. Springer Nature, Singapore (2024). https://doi.org/10.1007/978-981-99-8413-8_4
6. Biggio, L., Bendinelli, T., Neitz, A., Lucchi, A., Parascandolo, G.: Neural symbolic regression that scales. In: *International Conference on Machine Learning*, pp. 936–945. PMLR (2021)
7. Brunton, S.L., Proctor, J.L., Kutz, J.N.: Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proc. Natl. Acad. Sci.* **113**(15), 3932–3937 (2016)
8. Da Costa, L., Landry, J.A., Levasseur, Y.: Treating noisy data sets with relaxed genetic programming. In: *Artificial Evolution: 8th International Conference, Evolution Artificielle, EA 2007, Tours, France, October 29–31, 2007, Revised Selected Papers*, pp. 1–12. Springer, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-540-79305-2_1
9. Efron, B., Hastie, T., Johnstone, I., Tibshirani, R.: Least angle regression. *Ann. Stat.* **32**(2), 407–499 (2004). <https://doi.org/10.1214/009053604000000067>
10. Foret, P., Kleiner, A., Mobahi, H., Neyshabur, B.: Sharpness-aware minimization for efficiently improving generalization. In: *ICLR Spotlight* (2021). <https://arxiv.org/abs/2010.01412>

11. Gonçalves, I., Silva, S., Fonseca, C.M.: On the generalization ability of geometric semantic genetic programming. In: Machado, P., Heywood, M.I., McDermott, J., Castelli, M., García-Sánchez, P., Burelli, P., Risi, S., Sim, K. (eds.) *Genetic Programming*, pp. 41–52. Springer International Publishing, Cham (2015)
12. Gonçalves, I., Silva, S., Fonseca, C.M., Castelli, M.: Unsure when to stop? In: *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM (2017). <https://doi.org/10.1145/3071178.3071328>
13. Gonçalves, I., Silva, S., Melo, J.B., Carreiras, J.M.B.: Random sampling technique for over-fitting control in genetic programming. In: Moraglio, A., Silva, S., Krawiec, K., Machado, P., Cotta, C. (eds.) *Genetic Programming*, pp. 218–229. Springer, Berlin, Heidelberg (2012)
14. Harrison, D., Rubinfeld, D.L.: Hedonic housing prices and the demand for clean air. *J. Environ. Econ. Manag.* **5**(1), 81–102 (1978). <https://www.sciencedirect.com/science/article/pii/0095069678900062>
15. Haut, N., Banzhaf, W., Punch, B.: Correlation versus RMSE loss functions in symbolic regression tasks. In: Trujillo, L., Winkler, S.M., Silva, S., Banzhaf, W. (eds.) *Genetic Programming Theory and Practice XIX*, pp. 31–55. Springer Nature, Singapore (2023). https://doi.org/10.1007/978-981-19-8460-0_2
16. Jin, Y., Branke, J.: Evolutionary optimization in uncertain environments—a survey. *IEEE Trans. Evol. Comput.* **9**(3), 303–317 (2005)
17. Keijzer, M.: Improving symbolic regression with interval arithmetic and linear scaling. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., Costa, E. (eds.) *Genetic Programming*, pp. 70–82. Springer, Berlin, Heidelberg (2003)
18. Koza, J.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press (1992)
19. Koza, J.R.: Genetic programming as a means for programming computers by natural selection. *Stat. Comput.* **4**, 87–112 (1994)
20. La Cava, W., Orzechowski, P., Burlacu, B., de Franca, F., Virgolin, M., Jin, Y., Kommenda, M., Moore, J.: Contemporary symbolic regression methods and their relative performance. In: Vanschoren, J., Yeung, S. (eds.) *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, vol. 1 (2021). https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/c0c7c76d30bd3dcaefc96f40275bdc0a-Paper-round1.pdf
21. Li, H., Xu, Z., Taylor, G., Studer, C., Goldstein, T.: Visualizing the loss landscape of neural nets. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18*, pp. 6391–6401. Curran Associates Inc., Red Hook, NY, USA (2018)
22. Livadiotis, G., McComas, D.J.: Fitting method based on correlation maximization: applications in space physics. *J. Geophys. Res.: Space Phys.* **118**(6), 2863–2875 (2013). <https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/jgra.50304>
23. Luke, S., Panait, L.: Fighting bloat with nonparametric parsimony pressure. In: Guervós, J.J.M., Adamidis, P., Beyer, H.G., Schwefel, H.P., Fernández-Villacañes, J.L. (eds.) *Parallel Problem Solving from Nature - PPSN VII*, pp. 411–421. Springer, Berlin, Heidelberg (2002)
24. Moraglio, A., Krawiec, K., Johnson, C.: Geometric semantic genetic programming. In: Coello, C., Cutello, V., Deb, K., Forrest, S., Nicosia, G., Pavone, M. (eds.) *Parallel Problem Solving from Nature - PPSN XII. Lecture Notes in Computer Science*, vol. 7491, pp. 21–31. Springer, Berlin Heidelberg (2012)
25. Ni, J., Rockett, P.: Tikhonov regularization as a complexity measure in multiobjective genetic programming. *IEEE Trans. Evol. Comput.* **19**(2), 157–166 (2015). <https://doi.org/10.1109/TEVC.2014.2306994>
26. Nordin, P., Banzhaf, W.: An on-line method to evolve behavior and to control a miniature robot in real time with genetic programming. *Adapt. Behav.* **5**(2), 107–140 (1997)
27. Pietropolli, G., Manzoni, L., Paoletti, A., Castelli, M.: Combining geometric semantic GP with gradient-descent optimization. In: *Genetic Programming: 25th European Conference. EuroGP 2022, Held as Part of EvoStar 2022, Madrid, Spain, April 20–22, 2022, Proceedings*, pp. 19–33. Springer, Berlin, Heidelberg (2022)

28. Pietropolli, G., Manzoni, L., Paoletti, A., Castelli, M.: On the hybridization of geometric semantic GP with gradient-based optimizers. *Genet. Program. Evol. Mach.* **24**(2), Article number: 16 (2023). <https://rdcu.be/dpWsR>. Online first
29. Rakshit, P., Konar, A., Das, S.: Noisy evolutionary optimization algorithms-a comprehensive survey. *Swarm Evol. Comput.* **33**, 18–45 (2017)
30. Silva, S., Costa, E.: Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. *Genet. Program. Evol. Mach.* **10**, 141–179 (2009). <https://api.semanticscholar.org/CorpusID:10925054>
31. Sudholt, D.: Analysing the robustness of evolutionary algorithms to noise: refined runtime bounds and an example where noise is beneficial. *Algorithmica* **83**(4), 976–1011 (2021). <https://doi.org/10.1007/s00453-020-00671-0>
32. Topchy, A., Punch, W.F.: Faster genetic programming based on local gradient search of numeric leaf values. In: *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation, GECCO'01*, pp. 155–162. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)
33. Tsanas, A., Xifara, A.: Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools. *Energy Build.* **49**, 560–567 (2012). <https://www.sciencedirect.com/science/article/pii/S037877881200151X>
34. Vanneschi, L.: An introduction to geometric semantic genetic programming. In: Schütze, O., Trujillo, L., Legrand, P., Maldonado, Y. (eds.) *NEO 2015: Results of the Numerical and Evolutionary Optimization Workshop NEO 2015 held at September 23–25 2015 in Tijuana, Mexico*, pp. 3–42. Springer International Publishing, Cham (2017)
35. Vanneschi, L., Castelli, M., Silva, S.: Measuring bloat, overfitting and functional complexity in genetic programming. In: *Proceedings of the 12th annual Conference on Genetic and Evolutionary Computation*, pp. 877–884 (2010)
36. Vanneschi, L., Gustafson, S.: Using crossover based similarity measure to improve genetic programming generalization ability. In: *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, pp. 1139–1146 (2009)
37. Vanneschi, L., Silva, S., Castelli, M., Manzoni, L.: Geometric semantic genetic programming for real life applications. In: Riolo, R., Moore, J.H., Kotanchek, M. (eds.) *Genetic Programming Theory and Practice XI*, pp. 191–209. Springer, New York, NY (2014)
38. Vladislavleva, E.J., Smits, G.F., den Hertog, D.: Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *IEEE Trans. Evol. Comput.* **13**(2), 333–349 (2009)
39. Yeh, I.C.: Concrete compressive strength. UCI machine learning repository (2007). <https://doi.org/10.24432/C5PK67>
40. Zhang, H., Chen, Q., Xue, B., Banzhaf, W., Zhang, M.: Sharpness-aware minimization for evolutionary feature construction in regression (2024). arXiv: <https://arxiv.org/pdf/2405.06869>

Chapter 9

Tree-Based Grammatical Evolution with Non-Encoding Nodes



Marina de la Cruz, Oscar Garnica, J. Manuel Velasco, Daniel Parra,
and J. Ignacio Hidalgo

Abstract Grammar-guided genetic programming is a type of genetic programming that uses grammatical rules to constrain the search space of the solutions. There are different representations of grammar-guided genetic programming, including grammatical evolution, structured grammatical evolution, and context-free grammar genetic programming. Each representation introduces distinctive characteristics that shape the evolutionary process. In this chapter, we propose a new representation that uses a tree structure with non-encoding nodes for the individuals in the population; Tree-Based Grammatical Evolution with non-encoding Nodes. This representation increases the size and complexity of the individuals while performing a more exhaustive exploration of the solution space. Each node of the tree will have a set of children nodes and an associated number; this number determines which of the children nodes are used in decoding the solution and which are non-encoding nodes. Those nodes carry out a concomitant evolutionary process that may manifest on the phenotype after eventual mutation or crossover events. We investigate the performance of our proposal with a set of well-known benchmarks. Experimental results analyze the differences and similarities with other methods that also use grammars to guide the solutions and indicate the utility of the tree-based grammatical evolution with non-encoding nodes.

Keywords Grammar-guided genetic programming · Dynamically structured grammatical evolution · Context-free grammar genetic programming · Grammatical evolution

9.1 Introduction

Grammar-Guided Genetic Programming (GGGP) is a variant of genetic programming (GP) that employs a grammar to define the structural composition of individuals within the population. The use of grammars in the construction of the solutions allows

M. de la Cruz · O. Garnica · J. M. Velasco · D. Parra · J. I. Hidalgo (✉)
Department of Computer Architecture, Universidad Complutense de Madrid, Madrid, Spain
e-mail: hidalgo@ucm.es; absys@ucm.es

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2025
S. M. Winkler et al. (eds.), *Genetic Programming Theory and Practice XXI*, Genetic and Evolutionary Computation, https://doi.org/10.1007/978-981-96-0077-9_9

177

the creation of individuals that contain expert knowledge for the specific problem and limits the accessible search space.

Different GGGP versions have been proposed, including Grammatical Evolution (GE) [7], which uses a linear representation of the genotype, Structured Grammatical Evolution (SGE) [10], which uses multiple linear representations, one for each rule of the grammar and, Context-Free Grammar GP (CFG-GP) [22], where the genotype is a derivation tree generated from the grammar. Each representation has a different effect on the evolutionary process, with its advantages and drawbacks.

9.1.1 Context-Free Grammars

Context-free grammars (CFG) are formal grammars defined by Chomsky [6] that have been applied in various contexts, including GP. Formally, a CFG is defined as a tuple $G = (N, T, S, P)$, where N corresponds to a set of non-terminal symbols, T is the set of terminal symbols, S is the rule that begins the grammar, called axiom, and P the set of productions for each of the rules. The productions adhere to the form $A ::= B$, with A belonging to N , and B any sequence of terminals and non-terminals belonging to N and T , respectively, ($B \in (N \cup T)^*$). Each non-terminal can have multiple productions separated by the $|$ operator. This syntax, known as the Backus–Naur Form, will be employed throughout the chapter to define the grammar used to generate solutions. We will illustrate the GGGP techniques described in this chapter with the grammar of Fig. 9.1.

```
# Model expression
<expr> ::= <expr> <op> <expr> | <pre_op> <expr> | <var>

#Variables
<var> ::= x1 | x2 | x3 | x4

#Mathematical operators
<op>   ::= + | -
<pre_op> ::= log() | sin() | cos()
```

Fig. 9.1 Grammar used in BNF format

This grammar allows the GP algorithm to construct mathematical expressions in a recursive manner. It permits the inclusion (or not) of four variables (x_1 , x_2 , x_3 , and x_4) as many times as necessary and operated (+ or −) or preoperated (by log, sin, and cos).

9.1.2 Grammatical Evolution

Grammatical Evolution (GE) [7] is a linear representation of a GGGP. In GE, the genotype of the individuals within the population is structured as a list of numerical values with a predetermined size. These numerical values can be expressed either as integers or as 8-bit binary numbers. They are initially generated randomly within the range of 0 to a specified maximum, typically set to 128 or 256. These numbers are used in the decoding process aimed at deriving phenotypes from the genotype.

The decoding process starts with the interpretation of the first non-terminal symbol specified by the grammar. This interpretation uses the first number of the genotype to obtain a production from the grammar rule, performing the modulus operator on the number of productions associated with the rule. Subsequently, it traverses the symbols of the selected production. Terminals are appended directly to the evolving phenotype, while non-terminals prompt the recursive application of the decoding process with subsequent numbers from the genotype list. In essence, the decoding process generates a derivation tree, traversed in pre-order, whose structure is determined by the genotype within the language specified by the CFG. The leaves of the derivation tree collectively constitute the phenotype. The phenotype will be a mathematical expression in a symbolic regression problem, a derivation tree in a classification problem, or a computer program in a classical GP problem.

An example of a possible genotype structure, which can be derived from the grammar in Fig. 9.1, is illustrated in Eq. (9.1). It generates the phenotype $x_1 + \sin(x_3)$.

$$[3, 2, 8, 4, 1, 7, 5, 14] \xrightarrow{\text{decode}} x_1 + \sin(x_3) \quad (9.1)$$

GE has some issues derived from its representation. These are uncontrolled locality and redundancy in gene operations. The concept of low locality is the notion that small changes in the genotype will likewise induce subtle changes in the phenotype. However, in the case of GE, we find that this is only sometimes the case, as performing small changes in the genotype may yield a substantially altered derivation tree during the decoding process. Additionally, redundancy emerges as another significant issue within GE. Most of the changes performed on the genotype have been shown to not produce a change on the phenotype [19]. This phenomenon arises from the use of the modulus operator, as the same production can be obtained from different numbers, resulting in the same phenotype.

The main advantage of GE, in comparison to other methods, is the decoupling of the genotype from the phenotype. It allows one individual to be used in different contexts and with different grammar structures without having to make any modifications.

9.1.3 Structured Grammatical Evolution

In lieu of GE's locality and redundancy problems, Lourenço et al. proposed a novel approach termed Structured Grammatical Evolution (SGE) [10]. SGE diverges from traditional GE in the genotype representation; instead of a list of numbers, SGE employs a list of lists, where each internal list corresponds to a non-terminal in the grammar. The decoding process is similar to GE, albeit with the incorporation of individual lists corresponding to each non-terminal symbol within the grammar. Decoding a specific non-terminal involves first, selecting the list that represents that symbol, second selecting the next unused number from that list, and finally, the number is used to get the next production from the grammar. The adoption of separate lists for each rule nullifies the need to use the modulus operator. In each internal list, the numbers are between 0 and $n - 1$, with n being the total number of productions of the non-terminal symbol in question.

An example of a possible genotype structure that can be derived from the grammar of Sect. 9.1.1 is illustrated in Eq. (9.2). It generates the phenotype $x1 + \sin(x3)$.

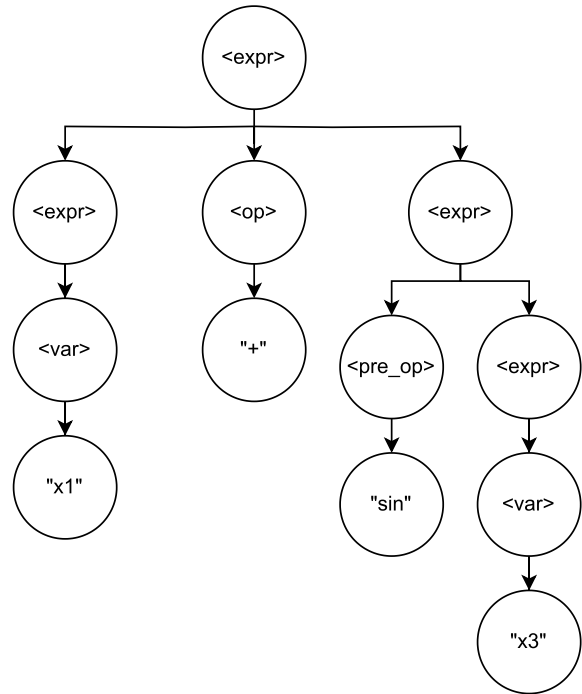
$$\left[\begin{array}{c} \underbrace{\text{<expr>}}_{[0, 2, 1, 2]}, \underbrace{\text{<expr>}}_{[0, 2]}, \underbrace{\text{<op>}}_{[0]}, \underbrace{\text{<pre_op>}}_{[1]} \end{array} \right] \xrightarrow{\text{decode}} x1 + \sin(x3) \quad (9.2)$$

SGE eliminates the need for the modulus operator, alleviating the redundancy issues found in GE. Additionally, the internal lists facilitate crossover operations at a finer granularity, producing children individuals with phenotypes more akin to their parents. Two versions of SGE have been proposed: static [11] and dynamic [10]. In this discourse, our focus will be on dynamic SGE, characterized by the dynamic generation of internal lists for each non-terminal symbol. During the initialization phase, the lists are generated up to what the specific individual needs to complete the phenotype generation and set the fitness. Following mutation and crossover operations, if more elements are needed for a rule to complete the phenotype generation, they are appended at the end of the internal list. This dynamic list management strategy ensures that only valid individuals exist in the population. Since the lists are dynamic, a maximum tree depth is imposed to mitigate potential bloating issues when generating recursive productions.

9.1.4 Context-Free Grammar Genetic Programming

In tree-based GGGP, individuals are derivation trees whose creation is guided by the grammar. Context-free grammar Genetic Programming (CFG-GP) is a version that uses a CFG to create the derivation trees. It was proposed by Whigham [22], and then several versions were developed. In this chapter, we adopt the original implementation of CFG-GP. The tree comprises nodes, each containing a symbol and pointers

Fig. 9.2 Example genotype context-free grammar GP



to child nodes that collectively compose the tree structure. The structure of the tree is dictated by the grammar file, with the root node corresponding to the first non-terminal symbol specified in the grammar. Subsequent child nodes are determined by the production chosen by the individual tree. In CFG-GP, there is a one-to-one relationship between phenotype and genotype. Decoding entails traversing the tree in a pre-order fashion, extracting the values of leaf nodes, which contain terminal symbols, and generating the phenotype. Figure 9.2 presents how the genotype generates the phenotype $x1 + \sin(x3)$ using the grammar in Fig. 9.1.

In CFG-GP, the derivation trees contain a possible structure that the grammar can generate by choosing a certain production for each non-terminal symbol. A random generation of trees might lead to an uneven population, particularly with complex grammars. Hence, different initialization techniques have been proposed to obtain a more uniform initial population [3, 8].

9.2 Tree-Based Grammatical Evolution with Non-Encoding Nodes (T-GE-NEN)

We propose a novel form of GGPP that combines features of CFG-GP and GE, namely Tree-Based Grammatical Evolution with Non-Encoding Nodes (T-GE-NEN). Central to our proposal is the use of non-encoding nodes, i.e., nodes that, while not directly impacting the phenotype, carry out a concomitant evolutionary process that may manifest on the phenotype after eventual mutation or crossover events. These nodes preserve additional information about the grammar structure, so the evolutionary operators are not as destructive as other GGPP versions.

T-GE-NEN adopts a tree-based representation, where each node corresponds to a non-terminal symbol in the grammar. Algorithm 9.1 defines the attributes of each node: the non-terminal symbol governing the associated rule, a numerical identifier determining the selection of the rule's production, and the collection of potential child nodes representing subsequent non-terminal symbols.

Algorithm 9.1 Node definition

```

1: Node attributes
2:   symbol  $\leftarrow$  Symbol()
3:   production  $\leftarrow$  Integer()
4:   childrenNodes  $\leftarrow$  List[Node]
5: end Node attributes

```

In contrast to a traditional derivation tree, the set of children nodes may contain both encoding and non-encoding nodes. Whether a node is encoding is dependent on the production generated by its parent node. Figure 9.3 depicts a segment of a tree, exemplifying this concept.

Let us assume that the first digit in the genotype is 1. The upper node in the figure denotes the `<expr>` non-terminal symbol, with the number 1 selecting the second production of the right-hand side of the rule associated with `<expr>`, producing `<expr> :: = <pre_op> <expr>`. Dividing this production into its non-terminal elements results in `<pre_op>` and `<expr>`. Both types of nodes will be present in the list of child nodes of the parent `<expr>`.

Being 1 the actual genotype, only `<pre_op> <expr>` is active in the current generation. However, if, during evolution, alterations occur in the numerical assignment associated with the parent (e.g., from 1 to 0), the production would become `<expr> :: = <expr> <op> <expr>`.

This alternate production contains three non-terminal symbols. During the decoding process, T-GE-NEN would consider three children nodes: the first node of type `<expr>`, which is the same node that the previous production generated; the node of type `<op>`; and the second node of type `<expr>`. This configuration appears on the right-hand side of Fig. 9.3.

One of the attributes of T-GE-NEN is that the nodes are shared among productions when they have the same type and amount of non-terminals.

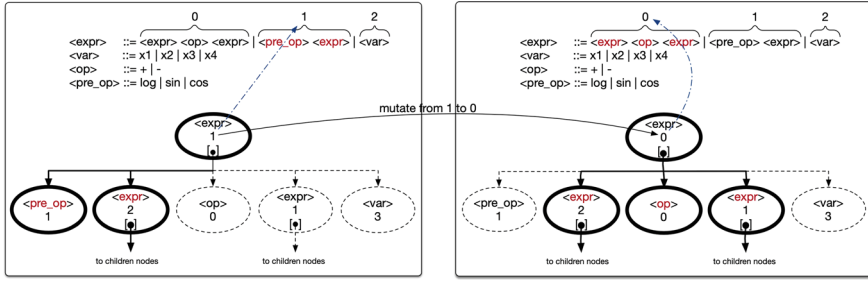


Fig. 9.3 T-GE-NEN mutation example

9.2.1 Initialization

The initialization procedure for the data structure involves generating expansions, following the grammar structure, and constructing the tree accordingly. Algorithm 9.2 defines the procedure. This process closely resembles initialization for CFG-GP, with the inclusion of a numerical value assigned to all nodes to indicate the generated production. Furthermore, as shown in Fig. 9.3, the terminal symbols do not generate nodes in the tree structure since the production number already represents them. The structure of all initial trees contains only encoding nodes; only the nodes used in the phenotype construction are generated. The rest of the structure will develop throughout the evolutionary process. The initialization process can be performed by generating random expansions until the entire tree has been generated or performing more complex generation techniques such as PTC2 [8, 12]. In Algorithm 9.2, line 3 parameterizes the generation of the next expansion.

Algorithm 9.2 Initialization process

```

1: procedure INIT(node, depth, grammar, maxDepth)
2:   rule  $\leftarrow$  grammar.FINDRULE(node.symbol)
3:   node.production  $\leftarrow$  GENERATEPRODUCTION(rule)
4:   for all symbol  $\in$  rule.GET(node.production) do
5:     if symbol.ISNOTTERMINAL() then
6:       childrenNodes  $\leftarrow$  []
7:       production  $\leftarrow$  null
8:       newNode  $\leftarrow$  node.NEW(symbol, production, childrenNodes)
9:       INIT(newNode, depth+1, grammar, maxDepth)
10:    end if
11:  end for
12: end procedure

```

The basic structure is initiated by identifying the rule of the grammar associated with the symbol passed to the function. The next production is determined through the GENERATEPRODUCTION(rule) call. Afterward, the production is set as the value

of the current node. Then, the chosen production is iterated over, and all non-terminal symbols of the production create new nodes. Such nodes are generated and added to the set of children nodes of the parent through the *node.NEW(symbol,value, childrenNodes)* function where *node* is the parent node. The call creates a node whose type is the next non-terminal symbol of the production, with *childrenNodes* as an empty list of type *List[Node]* and an undefined value. This, initially, null number is modified recursively when the production number associated with the node is defined at *GENERATEPRODUCTION*. The function is called recursively with the new node, thus constructing the tree.

9.2.2 Decoding

Algorithm 9.3 describes the process used for transforming the genotype to the phenotype. The procedure assumes the dual task of validating the integrity of the genotype and compliance with the imposed size constraints. We have chosen to fuse both functionalities for efficiency reasons, so the number of times the tree is completely traversed is as small as possible. The decoding procedure obtains the grammar rule from the non-terminal symbol of the node and the production it generates. If the branch goes over the maximum depth, and the rule is recursive with the next production, evaluated with the function *ISRECURSIVEPRODUCTION(rule, production)*, it modifies the production generated to become the shortest path to a terminal.

Algorithm 9.3 Decoding and revising procedure

```

1: procedure DECODING(node, maxDepth, phenotype, grammar, depth)
2:   rule  $\leftarrow$  grammar.FINDRULE(node.symbol)
3:   production  $\leftarrow$  rule.GET(node.production)
4:
5:   if depth  $\geq$  maxDepth && ISRECURSIVEPRODUCTION(rule, production) then
6:     CHANGERECURSION(node, grammar)
7:     production  $\leftarrow$  rule.GET(node.production)
8:   end if
9:   for all symbol  $\in$  production do
10:    if symbol.ISTERMINAL() then
11:      phenotype.APPEND(symbol)
12:    else
13:      nextNode  $\leftarrow$  node.NEXTCHILD(symbol)
14:      if nextNode == NULL then  $\triangleright$  Current node has no children nodes for the symbol
15:        childrenNodes  $\leftarrow$  []
16:        randomVal  $\leftarrow$  GENERATEPRODUCTION(symbol)
17:        newNode  $\leftarrow$  node.NEW(symbol, randomVal, childrenNodes)
18:      end if
19:      DECODING(newNode, maxDepth, phenotype, grammar, depth+1)
20:    end if
21:  end for
22: end procedure

```

The algorithm then iterates over each symbol of the production. If the next symbol is terminal, it adds it to the construction phenotype. Otherwise, it retrieves the appropriate child node from a given grammar symbol through the procedure *node.NEXTCHILD(symbol)*. This procedure should return the next unused child node that contains the defined symbol. In cases where no unused node exists for the request, a null value is returned. In Algorithm 9.3, when it receives a null value instead of a valid node, the sub-tree is generated randomly, forcing the completion of the phenotype. This is performed through the *GENERATEPRODUCTION(symbol)* function, which returns a valid random value within the productions of the following rule *symbol*. The *node.NEW(symbol, randomVal, childrenNodes)* instruction has the same function as in the initialization procedure; it creates a new node and associates it with the parent. This may occur when, through the evolutionary process, a node changes the generated production, and the new production contains a symbol not present in the previous one. This is how the set of children nodes is generated dynamically through the evolutionary process. The depth restriction ensures that the newly generated nodes adhere to the depth limit set.

9.2.3 Crossover Operator

Based on the structure of the genotype tree, the crossover operator is the general sub-tree crossover. The operator selects a non-terminal symbol in both parent solutions and exchanges the corresponding instances between the trees. Given the presence of both encoding and non-encoding nodes within the trees, specific constraints have been added to determine the type of nodes eligible for crossover. Algorithm 9.4 shows the constraints employed in our study. They dictate that 25% of the time, the selected node in both trees is encoding; 50% of the time, the node from one of the trees, selected at random, is chosen from all nodes, including the non-encoding ones; and the remaining 25% of the time the nodes from both trees are chosen randomly out of every now in the tree. Consequently, the probability of selecting a non-encoding node increases if the tree contains a higher density of these types of nodes. The constraints proposed are designed to ensure that at least 75% of the time, one of the final individuals undergoes a phenotype change. However, the specified rates are adjustable; for some problems, it may yield better results to allow more or less freedom in the node selection process. The procedure is described in Algorithm 9.4 by generating two random Boolean values. Based on the results, a random encoding node may be retrieved, *GETRANDOMENCODINGNODE(ind, symbol)*, or a random node from all available, *GETRANDOMNODE(ind, symbol)*.

In line 18, the swap of the tree nodes includes all the children nodes, both encoding and non-encoding. Figure 9.4 presents an example of the described procedure. An alternate version of the crossover could consist of only the encoding children nodes of a sub-tree so that only the phenotypical part is swapped between the trees. The non-encoding nodes would remain in their original tree. This alternate version has been developed but will not be used in the results section.

Algorithm 9.4 Crossover procedure

```

1: procedure SUBTREE_CROSSOVER(ind1, ind2, grammar)
2:   bool1 ← UNIFORMRANDBOOLEAN()
3:   bool2 ← UNIFORMRANDBOOLEAN()
4:   symbol ← GETRANDOMSYMBOL(grammar)
5:   if bool1 && bool2 then
6:     node1 ← GETRANDOMENCODINGNODE(ind1, symbol)
7:     node2 ← GETRANDOMENCODINGNODE(ind2, symbol)
8:   else if bool1 then
9:     node1 ← GETRANDOMENCODINGNODE(ind1, symbol)
10:    node2 ← GETRANDOMNODE(ind2, symbol)
11:   else if bool2 then
12:    node1 ← GETRANDOMNODE(ind1, symbol)
13:    node2 ← GETRANDOMENCODINGNODE(ind2, symbol)
14:   else
15:    node1 ← GETRANDOMNODE(ind1, symbol)
16:    node2 ← GETRANDOMNODE(ind2, symbol)
17:   end if
18:   (node1, node2) ← (node2, node1)
19: end procedure

```

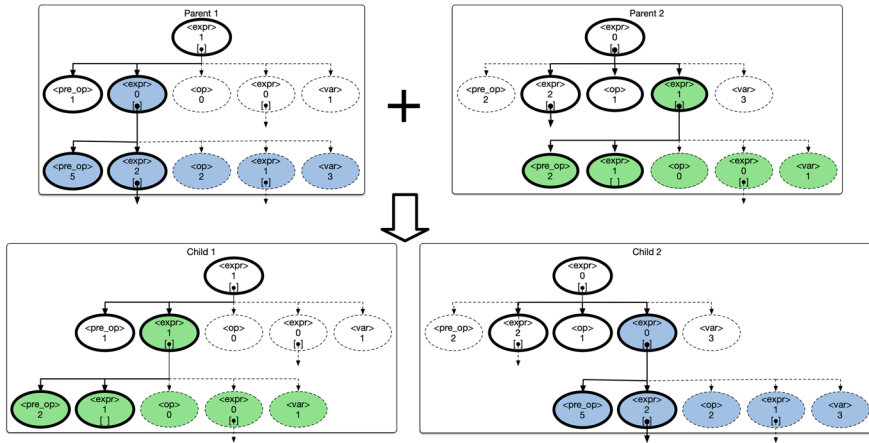


Fig. 9.4 T-GE-NEN crossover example. Dotted line circles represent non-encoding nodes, while solid-lined circles are the active (encoding) nodes. Green and blue colors highlight the parts of each parent participating in the crossover operation

9.2.4 Mutation Operator

Mutation targets the attribute that determines the production encoded by each node. If a node is selected for mutation, a random number between 0 and the total number of productions of the non-terminal symbol will be drawn. The value of the node is swapped by the newly drawn number.

Similar to the approach adopted in the crossover, a set of constraints is incorporated into the mutation operator to account for the underlying structure. The selection procedure to determine which nodes are fit for mutation is divided between choosing from the encoding nodes and choosing from every tree node. A random value is drawn; if it is below a specific threshold, the node is chosen from the encoding nodes, or else it is drawn from every node of the tree, at which point the probability of it being encoded depends on the percentage of encoding vs non-encoding nodes. The threshold chosen for the study is set to 75%.

9.2.5 Bloating Control

In T-GE-NEN, the issue of bloating is a significant concern, requiring control measures at both the encoding and non-encoding nodes. These measures can be performed independently. That is, a maximum tree depth can be established for the encoding nodes and a different tree depth assigned for the non-encoding nodes, provided that the overall tree sizes remain within a reasonable size. For the encoding portion, the control is performed through Algorithm 9.3.

The non-encoding nodes, on the other hand, are limited using a very similar technique, shown in Algorithm 9.5. The bloating control is performed at the end of each generation; once the individuals have gone through mutation and crossover, it is performed through a depth limit. If a tree branch goes over the depth limit and the rule symbol is recursive, then all recursive children's nodes are deleted. This is performed through the `DELETERECURSIVENODES(node, rule)` instruction, which simply deletes any children nodes of recursive nodes. If the next production the rule generates is recursive with the rule, `ISRECURSIVE(rule, production)`, the production is modified to generate the shortest path to a terminal.

Algorithm 9.5 Bloating control for non-encoding nodes

```

1: procedure CUTTREE(node, maxDepth, grammar, depth)
2:   rule  $\leftarrow$  grammar.FINDRULE(node.symbol)
3:   production  $\leftarrow$  nextRule.GET(node.production)
4:
5:   if depth  $\geq$  maxDepth && ISRECURSIVE(rule) then
6:     if ISRECURSIVEPRODUCTION(rule, production) then
7:       CHANGERECURSION(node, rule)
8:     end if
9:     DELETERECURSIVENODES(node, rule)
10:  end if
11:  for all childNode  $\in$  node.childrenNodes do
12:    CUTTREE(childNode, maxDepth, grammar, depth+1)
13:  end for
14: end procedure

```

9.2.6 Other Approaches with Non-encoding Genes

We can find in the literature some previous GP approaches where non-encoding genes participate in the evolutionary process. For example, stack-based genetic programming [15]. Perkins proposed a GP approach that utilizes an execution model that uses a stack-based virtual machine to evaluate GP individuals. Non-encoding nodes appear when the stack does not have enough values to perform an operation. For example, if an addition of two variables is indicated by the phenotype, but there is only one value on the stack. This is not a desired effect and produces non-valid individuals, so it is not comparable with the proposal presented in this chapter.

Non-encoding genes could also appear in Cartesian GP [13] under some conditions since not all genes in the genotype have to be active to decode a complete expression. As in T-GE-NEN, those genes pass genetic information through generations and genetic operators act on them too, experimenting a parallel *hidden evolution*. We explored also this alternative over hardware implementations [5]. As CGP uses a $\mu + \lambda$ evolutionary strategy algorithm [2], future work exploring T-GE-NEN under this selection scheme could be interesting.

9.3 Experimental Results

This section presents the methodology and experimental results obtained for the proposed algorithm.

9.3.1 Experimental Setup

Table 9.1 presents the experimental setup for conducting the tests. Parameters have been selected to ensure comparability across all algorithms, with values that work well on all of them. However, no studies have been performed on the efficiency of choosing specific parameters in relation to others other than the experimental results. Common to all of the algorithms is the number of generations, set to 1000, the size of the population, 200, and the number of runs, 100, for a total of 200000 evaluations. The selection and replacement methods, with tournament selection of 5 individuals and a replacement policy of the parents for the offspring with 2% elitism. PTC2 initialization is employed across all four algorithms.

In our GE implementation, the structure of individuals maintains a fixed size throughout the evolutionary process, in contrast to the other three implementations where the structure can grow. Although this characteristic reduces computation time, it also limits the generation process. To partially alleviate this issue, the initial size of the individuals and their expansions are doubled from 100 to 200.

Table 9.1 Experimental setup for comparison of GE, DSGE, CFG-GP and T-GE-NEN. SubTree* and One gene* are the evolutionary operators previously described

	GE	DSGE	CFG-GP	T-GE-NEN
N ^o runs	100			
Gen	1000			
Pop	200			
Initialization method	PTC2			
Max expansions	200	100	100	100
Selection	Tournament pressure 5			
Elitism	2%			
Crossover operator	Single point	Uniform (mask)	SubTree	SubTree* 50%
Crossover prob	90%			
Mutation operator	Integer-Flip	Integer-Flip	SubTree regeneration	One-gene* 75%
Mutation prob	1%	1%	100%	100%
Max initial depth	10	10	10	10
Max depth	–	10	10	10
Chromosome length	200 (2 Wraps)	–	–	–

9.3.2 Benchmark Problems

Table 9.2 contains the list of benchmark problems selected to perform the tests. Most of the problems have been taken from [23]. They are four real-world problems and four mathematical functions. Six are regression problems, while the other two are classification problems.

Figure 9.5 depicts the grammar structure employed for all regression problems. The non-terminal <var> is modified to contain the available variables specific to each problem. Correspondingly, Fig. 9.6 illustrates the grammar structure used on the classification problems. The “else if” text is exclusive to the multi-class problem. For binary classification, the grammar only contains the first “if” statement. *RMSE* is the fitness function used for the regression benchmarks, whereas $(1 - \text{Accuracy})$ is the fitness function of the classification benchmarks. The proposed grammars have been constructed to have a simple structure, especially on regression. Within the regression grammar, the sole recursive rule is <expr>, which, along with <term>, can generate non-encoding nodes in T-GE-NEN. In the classification grammar, both <expr> and <binexpr> are recursive.

Table 9.2 Benchmark problems

Name	Variables	Equation	Type	Training
Keijzer-6 [9]	1	$\sum_i^x \frac{1}{i}$	Regression	E[1, 50, 1]
Vladislavleva-4 [21]	5	$\frac{10}{5 + \sum_{i=1}^5 (x_i - 3)^2}$	Regression	U[0.05, 6.05, 1024]
Nguyen-7 [20]	1	$\log(x+1) + \log(x^2+1)$	Regression	U[-1, 5, 1000]
Pagie Polynomial [14]	2	$\frac{1}{1+x^{-4}} + \frac{1}{1+y^{-4}}$	Regression	E[-5, 5, 0.4]
Dow Chemical (normalized) [17]	57	–	Regression	747 points
Abalone [1, 24]	8	–	Classification (3 classes)	2506 points
Banknote [16]	4	–	Classification (2 classes)	1000 points
Tower (normalized) [18]	5	–	Regression	4071 points

```

# Model expression
<expr> ::= <expr> <op> <expr> | <pre_op> <expr> | <var> | <term>
# options for terminals
<term> ::= <var> | <number>

# Variables
<var> ::= x1 | x2 | x3 | ..... xn

# Mathematical operators
<op>  ::= + | - | * | \
<pre_op> ::= log() | sin() | cos() | exp() | inv() | sqrt() | tan()

# Constants generation
<number> ::= <base> pow(10, <sign><exponent>)
<base>  ::= 1|2|3|4| ..... |99
<exponent> ::= 1|2|3|4|5|6|8|9
<sign>  ::= +|-

```

Fig. 9.5 Grammar for regression used in all representations

9.3.3 Results

Table 9.3 displays the Mean, Median, Std. Dev., and Min. Value obtained during the training phase across all four algorithms. For all eight problem instances, the Mean values attained during training by T-GE-NEN were the best. The Median obtains lower results in all problems except for Nguyen-7. T-GE-NEN obtains the best overall result for six out of the eight benchmarks, with CFG-GP finding a lower value on

```

# Model expression
<type> ::= if( <binexpr> ){class=0:}else{class=1:}
        else if( <binexpr> ){class=2:}
        .....
        else( <binexpr> ){class=n:}
# options for binary expressions
<binexpr> ::= ( <expr> <relop> <expr> ) |
              ( ( <expr> <relop> <expr> ) <binop> <binexpr> )

# Binary operators
<binop> ::= && | "||"
<relop> ::= "<" | ">" | "<=" | ">="

# Mathematical expressions
<expr> ::= ( <expr> <op> <expr> ) | <term> | <pre_op> <expr>

# Mathematical operators
<op> ::= + | - | * | \
<pre_op> ::= log() | sin() | cos() | exp() | inv() | sqrt() | tan()

# Variables
<var> ::= x1 | x2 | x3 | ... | xn

<term> ::= <var> | <number>
<var> ::= x1 | x2 | x3 | ... | xn
<op> ::= + | - | * | \

# Constants generation
<number> ::= <base> . pow(10,<sign><exponent>)
<base> ::= 1|2|3|4|...|99
<exponent> ::= 1|2|3|4|5|6|8|9
<sign> ::= +|-

```

Fig. 9.6 Grammar for decoding classification models as if-then-else expressions

the Pagie Polynomial and the Dow Chemical problem. In general, the algorithm proposed seems to find better training results than the other alternatives.

Figure 9.7 illustrates the evolutionary trajectories observed across all benchmark instances. Each plot showcases the evolutionary progression of the best individual over 100 runs conducted for each benchmark. The solid line depicts the average fitness value of the best individual across each generation, while the error band is the standard deviation of the best fitness values across all runs for each generation.

From these graphs, we can observe that, on average, T-GE-NEN consistently outperforms the other three algorithms throughout the evolutionary process. CFG-GP emerges as the next best-performing algorithm, followed by DSGE and GE. However, the disparity in performance between CFG-GP and T-GE-NEN does not seem too pronounced compared to the other two algorithms. This observation can be explained by the fact that both CFG-GP and T-GE-NEN use tree structures for the evolution procedures. This type of structure is more complex, requiring greater

Table 9.3 Mean, Median, Standard deviation, and Best Fitness for each algorithm and problem. Bold values represent the lowest value

Benchmark	Algorithm	Mean	Median	Std. dev	Min value
Pagie Polynomial	GE	0.391896	0.397145	0.093900	0.114032
Pagie Polynomial	DSGE	0.335472	0.330249	0.089490	0.082655
Pagie Polynomial	CFG-GP	0.222315	0.226208	0.076206	0.031809
Pagie Polynomial	T-GE-NEN	0.179255	0.186572	0.077647	0.032372
Tower	GE	0.106245	0.105829	0.017980	0.070854
Tower	DSGE	0.091207	0.089568	0.015240	0.066624
Tower	CFG-GP	0.078324	0.074586	0.014007	0.060275
Tower	T-GE-NEN	0.073374	0.070832	0.011196	0.060141
Vladislavleva-4	GE	0.193689	0.199103	0.017843	0.104983
Vladislavleva-4	DSGE	0.189788	0.189679	0.012619	0.152360
Vladislavleva-4	CFG-GP	0.143740	0.145498	0.027961	0.051027
Vladislavleva-4	T-GE-NEN	0.137415	0.143208	0.029619	0.044778
Dow Chemical	GE	0.115894	0.116878	0.019253	0.076254
Dow Chemical	DSGE	0.100412	0.098416	0.016260	0.066806
Dow Chemical	CFG-GP	0.081781	0.078189	0.013207	0.057188
Dow Chemical	T-GE-NEN	0.077076	0.074802	0.010935	0.060224
Abalone	GE	0.436077	0.435954	0.006069	0.418196
Abalone	DSGE	0.438344	0.438148	0.005766	0.423783
Abalone	CFG-GP	0.423412	0.425978	0.011943	0.393057
Abalone	T-GE-NEN	0.412678	0.414405	0.009245	0.389864
Keijzer-6	GE	0.045235	0.022517	0.054573	0.000471
Keijzer-6	DSGE	0.020618	0.007095	0.039168	0.000085
Keijzer-6	CFG-GP	0.002916	0.000289	0.009309	0.000009
Keijzer-6	T-GE-NEN	0.002273	0.000278	0.005629	0.000009
Nguyen-7	GE	0.264429	0.240511	0.095078	0.079676
Nguyen-7	DSGE	0.147206	0.136581	0.045373	0.065965
Nguyen-7	CFG-GP	0.076082	0.056020	0.047846	0.016678
Nguyen-7	T-GE-NEN	0.064248	0.057789	0.032874	0.013889
Banknote	GE	0.025770	0.021000	0.023478	0.000000
Banknote	DSGE	0.020850	0.019000	0.016235	0.000000
Banknote	CFG-GP	0.006380	0.000000	0.010288	0.000000
Banknote	T-GE-NEN	0.003010	0.000000	0.006583	0.000000

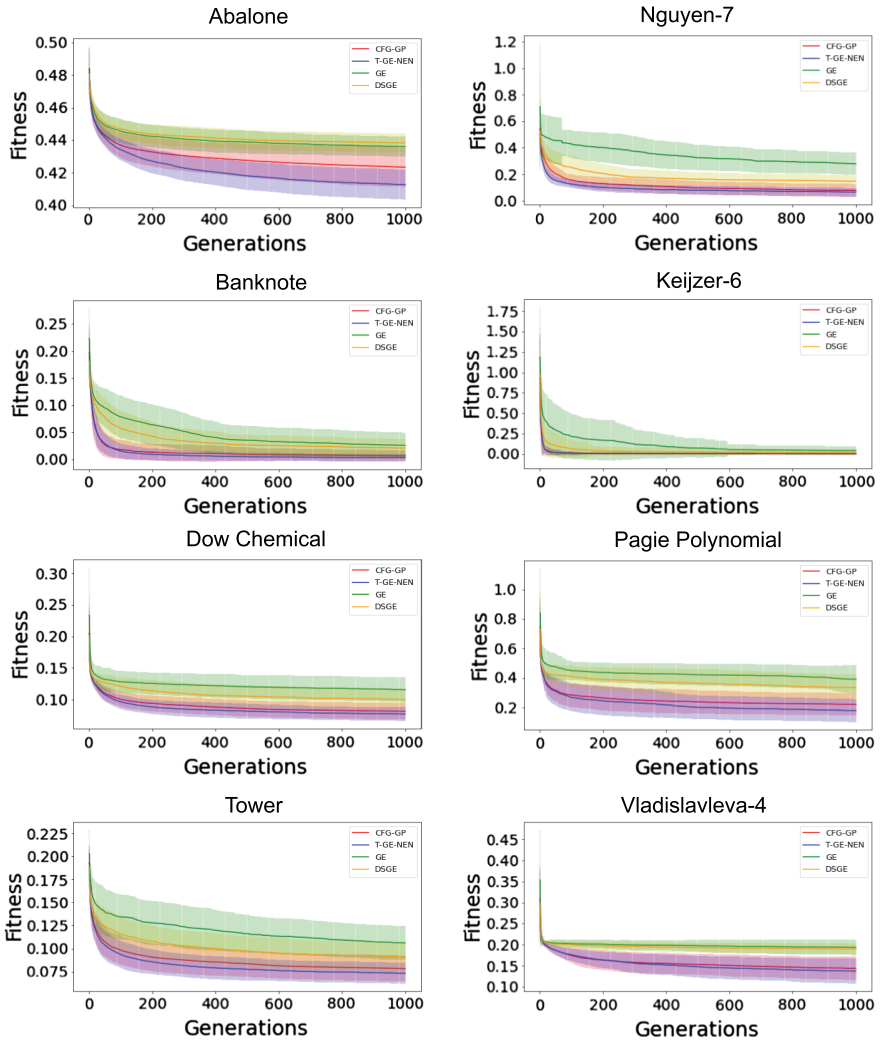


Fig. 9.7 Fitness generation evolution

computational resources and time, but the changes are always limited to the chosen portion of the tree. So, more granularity can be achieved, as observed in the results.

Regarding the best fitness, T-GE-NEN overtakes the other problems early in the evolutionary process, usually before generation 50. The Vladislavleva-4 problem is the only exception. On it, CFG-GP and T-GE-NEN exhibit comparable evolutionary trajectories until approximately generation 400, after which T-GE-NEN starts to yield visible superior performance.

On the Abalone problem, the proposed algorithm obtains significantly better results compared to the next best-performing CFG-GP. Additionally, the evolutionary

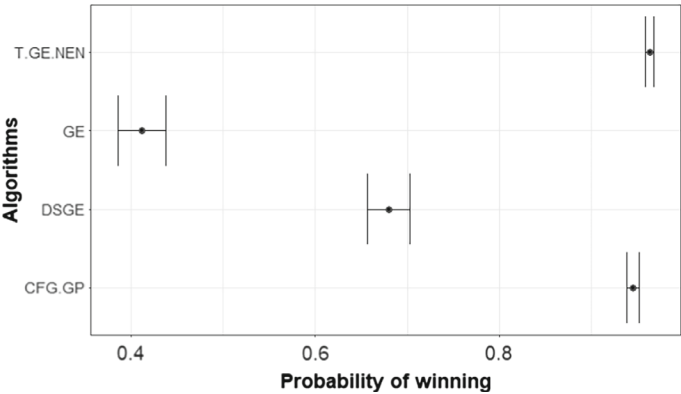
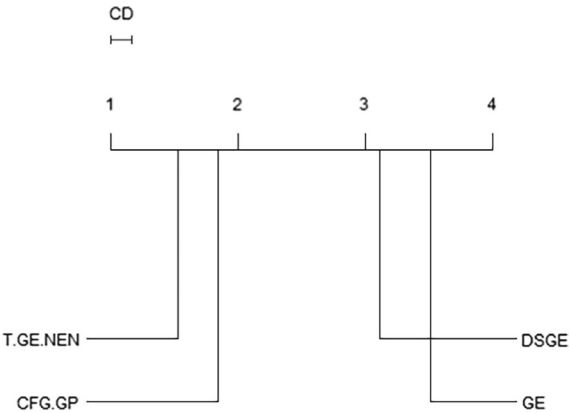


Fig. 9.8 Statistical analysis Bayesian ACC, Probability of winning

Fig. 9.9 Statistical analysis Bayesian CD ACC



curve for the Abalone problem exhibits a less pronounced elbow shape than the other benchmarks. We attribute this phenomenon to the complexity of the classification grammar, specifically for the multi-class problem.

Figures 9.8 and 9.9 present the outcomes derived from two techniques based on Bayesian models for comparing the methodologies. The Bayesian models are based on the Plackett-Luce distribution [4]. We use a significance level $\alpha = 0.05$, with 20 Monte Carlo chains and 4000 simulations. The best fitness results for the 100 runs of all algorithms are sent to the Bayesian models for a total of 800 data points. Figure 9.9 reveals a statistical significance in the results obtained for all four algorithms since there is no line connecting each of them, which would signify no statistical significance. Figure 9.8 depicts the probability of winning for each algorithm under this Bayesian model. On it, our proposal, T-GE-NEN, has a higher probability of being the best and has also slimmer bounds, making the result more robust.

9.4 Discussion

Several conclusions can be drawn from the presented results. Firstly, in the problems presented, the structure of the genotype clearly impacts the obtained results. Both tree representations, CFG-GP and T-GE-NEN yield better training values thanks to their more complex structures, which establish a one-to-one relationship between phenotype and genotype. Secondly, between T-GE-NEN and CFG-GP, our proposal, T-GE-NEN, obtained better results from the addition of non-encoding nodes to the tree structure. However, these non-encoding nodes increase the memory requirements of the genotype and increase the processing time for the evolutionary process, as computations must be performed on a larger number of nodes. The number of non-encoding nodes at any given time is mainly dependent on the tree depth and the grammar structure. More complex grammars, or those where productions don't share elements, will generate a greater number of non-encoding nodes. The amount is exponential to the maximum tree depth. Consequently, our proposal is generally more computationally expensive than the other options presented.

9.5 Conclusions and Future Work

In this chapter, we have introduced T-GE-NEN, a novel variant of GGGP featuring tree-based individuals comprising encoding and non-encoding nodes, together with information on the production of the grammar in use. The initial results obtained for the algorithm are encouraging, yet further development and comprehensive investigation are warranted to realize its full potential.

In terms of future work, a more exhaustive study of how the different versions of the evolutionary operators perform across different problem complexities is essential. The same should be done in terms of the crossover and mutation probabilities of the different versions. From the point of view of the tree structure, the amount of non-encoding nodes will be dependent on the structure of the grammar, affecting the performance gain of T-GE-NEN versus a normal derivation tree. To test whether this algorithm truly improves the performance, a study is needed on different grammar structures for the same problem, with differing amounts of non-encoding nodes. Furthermore, the algorithm could be evaluated on other problem domains not considered in this chapter, such as program synthesis. The performance of the algorithm should also be assessed using test datasets.

Acknowledgements This work has been supported by The Spanish Ministerio de Innovación Ciencia y Universidad-grants PID2021-125549OB-I00 and PDC2022-133429-I00.

References

1. archive.ics.uci.edu: Abalone dataset. <https://archive.ics.uci.edu/dataset/1/abalone>. Accessed 10 Mar. 2024
2. Beyer, H.G., Schwefel, H.P.: Evolution strategies-a comprehensive introduction. *Nat. Comput.* **1**, 3–52 (2002)
3. Böhm, W., Geyer-Schulz, A.: Exact uniform initialization for genetic programming. In: Belew, R.K., Vose, M.D. (eds.) *FOGA*, pp. 379–407. Morgan Kaufmann (1996). <http://dblp.uni-trier.de/db/conf/foga/foga1996.html#BohmG96>
4. Calvo, B., Santafé, G.: scmamp: Statistical comparison of multiple algorithms in multiple problems. *R J.* **8**, 1–8 (2016). <https://doi.org/10.32614/RJ-2016-017>
5. Cano, J., Hidalgo, J.I., Garnica, Ó.: Hardware real-time individualised blood glucose predictor generator based on grammars and Cartesian genetic programming. Preprint at Research Square (2024). <https://doi.org/10.21203/rs.3.rs-4262636/v1>
6. Chomsky, N., Schützenberger, M.: The algebraic theory of context-free languages ** this work was supported in part by the U.S. army signal corps, the air force office of scientific research, and the office of naval research; and in part by the national science foundation; and in part by a grant from the commonwealth fund. In: Braffort, P., Hirschberg, D. (eds.) *Computer Programming and Formal Systems. Studies in Logic and the Foundations of Mathematics*, vol. 26, pp. 118–161. Elsevier (1959). [https://doi.org/10.1016/S0049-237X\(09\)70104-1](https://doi.org/10.1016/S0049-237X(09)70104-1), <https://www.sciencedirect.com/science/article/pii/S0049237X09701041>
7. Dempsey, I., O'Neill, M., Brabazon, A.: *Grammatical Evolution*, pp. 9–24. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00314-1_2
8. García-Arnau, M., Manrique, D., Ríos, J., Rodríguez-Patón, A.: Initialization method for grammar-guided genetic programming. *Knowl.-Based Syst.* **20**(2), 127–133 (2007). <https://doi.org/10.1016/j.knosys.2006.11.006>, <https://www.sciencedirect.com/science/article/pii/S0950705106001973>. AI 2006
9. Keijzer, M.: Improving symbolic regression with interval arithmetic and linear scaling. In: *Genetic Programming*, pp. 70–82. Springer, Berlin, Heidelberg (2003). https://doi.org/10.1007/3-540-36599-0_7
10. Lourenço, N., Assunção, F., Pereira, F., Costa, E., Machado, P.: Structured Grammatical Evolution: A Dynamic Approach, pp. 137–161. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-78717-6_6
11. Lourenço, N., Pereira, F., Costa, E.: SGE: a structured representation for grammatical evolution. In: *Artificial Evolution*, pp. 136–148. Springer International Publishing (2015). https://doi.org/10.1007/978-3-319-31471-6_11
12. Luke, S.: Two fast tree-creation algorithms for genetic programming. *IEEE Trans. Evol. Comput.* **4**(3), 274–283 (2000). <https://doi.org/10.1109/4235.873237>
13. Miller, J.F., et al.: An empirical study of the efficiency of learning boolean functions using a Cartesian genetic programming approach. In: *Proceedings of the Genetic and Evolutionary Computation Conference*, vol. 2, pp. 1135–1142 (1999)
14. Pagie, L., Hogeweg, P.: Evolutionary consequences of coevolving targets. *Evol. Comput.* **5**(4), 401–418 (1997). <https://doi.org/10.1162/evco.1997.5.4.401>
15. Perkis, T.: Stack-based genetic programming. In: *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pp. 148–153. IEEE (1994)
16. PonyGE: Banknote normalized dataset (2017). <https://github.com/PonyGE/PonyGE2/blob/master/datasets/Banknote/Train.csv>. Accessed 5 Feb. 2024
17. PonyGE: Dow chemical normalized dataset (2017). <https://github.com/PonyGE/PonyGE2/blob/master/datasets/DowNorm/Train.txt>. Accessed 10 Feb. 2023
18. PonyGE: Tower normalized dataset (2017). <https://github.com/PonyGE/PonyGE2/blob/master/datasets/TowerNorm/Train.txt>. Accessed 10 Feb. 2023

19. Rothlauf, F., Oetzel, M.: On the locality of grammatical evolution. In: Genetic Programming, vol. 3905, pp. 320–330. Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/11729976_29
20. Uy, N.Q., Hoai, N.X., O'Neill, M., McKay, R.I., Galván-López, E.: Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genet. Program. Evol. Mach.* **12**(2), 91–119 (2011). <https://doi.org/10.1007/s10710-010-9121-2>
21. Vladislavleva, E.K., Smits, G., den Hertog, D.: Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *IEEE Trans. Evol. Comput.* **13**, 333–349 (2009). <https://doi.org/10.1109/TEVC.2008.926486>
22. Whigham, P.: Grammatically-based genetic programming. In: Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications (1999)
23. White, D.R., McDermott, J., Castelli, M., Manzoni, L., Goldman, B.W., Kronberger, G., Jaśkowski, W., O'Reilly, U.M., Luke, S.: Better GP benchmarks: community survey results and proposals. *Genet. Program. Evol. Mach.* **14**(1), 3–29 (2013). <https://doi.org/10.1007/s10710-012-9177-2>
24. Wilkinson, L., Anand, A., Tuan, D.N.: Chirp: a new classifier based on composite hypercubes on iterated random projections. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'11, pp. 6–14. Association for Computing Machinery, New York, NY, USA (2011). <https://doi.org/10.1145/2020408.2020418>

Chapter 10

Genetic Programming with Memory for Approximate Data Reconstruction



Lukas Sekanina and Tadeas Juza

Abstract This chapter addresses the computation–memorization trade-offs in the context of genetic programming (GP). We introduce genetic programming with memory (GPM) in which GP evolves not only the expression but also the content of a small local memory to better approximate the original data set. In particular, we evolved expression–memory pairs that can serve as weight generators and thus approximate the weights associated with convolutional layers of some convolutional neural networks (CNNs). This is potentially interesting for the efficient implementations of hardware accelerators of CNNs in which memory access is significantly more energy-demanding than arithmetic operations. In our approach, most of the weights are approximated using an evolved expression; only some fraction of them must be read from memory. For example, if memory contains 10% of the original weights, the weight generator evolved for a convolutional layer can approximate the original weights such that the CNN utilizing the generated weights shows less than a 1% drop in the classification accuracy on the MNIST data set. The memory requirements are reduced $3.1 \times$ or $12.6 \times$ for 8-bit or 32-bit weights, respectively. Additional experiments conducted for more complex CNNs and challenging image classification benchmarks show various impacts of weights' approximation on classification accuracy.

10.1 Introduction

Genetic programming (GP) is typically used to evolve an expression implementing an approximate mapping between two sets. Such an expression consists of input variables, operators, and constants. Since the constant values can be evolved, they can be seen as application-specific memory content needed to implement the mapping effectively. The concept of memory has been introduced into GP in different ways [8,

L. Sekanina (✉) · T. Juza

Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
e-mail: sekanina@fit.vutbr.cz

T. Juza

e-mail: tadeasjuza@gmail.com

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2025
S. M. Winkler et al. (eds.), *Genetic Programming Theory and Practice XXI*, Genetic and Evolutionary Computation, https://doi.org/10.1007/978-981-96-0077-9_10

199

9, 17, 18] but it has never been dominating in major applications of GP. Considering not only symbolic regression but computing in general, there are many cases in which it is more beneficial to memorize some knowledge rather than to compute it. For example, in hardware dividers or CORDIC algorithm [19], introducing a lookup table enables the development of fast and low-power implementations. This research addresses the **computation–memorization trade-offs** in the context of GP.

In our previous paper [7], we introduced *genetic programming with associative memory* (GPAM) in which GP evolves not only the expression but also the content of a small local memory to better approximate the original data set. Particularly, GPAM evolves an expression $G(x)$ and content of associative memory (content addressable memory) $AM(x)$ with the goal of minimizing the approximation error w.r.t. a chosen error metric. The resulting pair (G, AM) then produces the regression output y' such that if x is in AM then $y' = AM(x)$, otherwise $y' = G(x)$. GPAM was evaluated in two case studies.

Firstly, we considered symbolic regression for data sets containing many outliers. The goal is to memorize some of these outliers, as they could sometimes be useful. Unfortunately, they are almost always inexactly reproduced by any evolved expression in the standard GP approach. For this experiment, relevant synthetic data sets were obtained by sampling standard benchmark functions for symbolic regression in which some fraction of data points were replaced by randomly generated values.

Secondly, GPAM evolved an expression–memory pair to approximate a set of weights belonging to one layer of a convolutional neural network (CNN). The motivation was to replace energy-demanding reads of external weight memory (which a CNN hardware accelerator must conduct to perform every inference) with a cheap on-chip weight generator utilizing a small local memory. Promising results were obtained in a simplified setup that has not addressed some important aspects of the method, such as memory cost.

This work further analyzes and extends the GPAM approach in several directions. Since associative memory is sometimes unnecessary, we refer to the updated approach as *genetic programming with memory* (GPM). GPAM and GPM can employ any variant of GP. Like in [7], we will use Cartesian genetic programming (CGP) [12].

In the case of symbolic regression, we extend the number of benchmark problems to nine and evaluate them in a different setup. The original approach did not distinguish training and test data. Hence, we adopt a standard methodology of GP and perform the evolution with training data and the evaluation with test data; however, the randomly generated values (in the benchmark functions) remain identical in both sets.

In the case of the weight generator application, we propose CGP to work with 8-bit functions (in fixed-point representation) rather than 32-bit (floating-point) functions to allow for smaller hardware implementation of evolved expressions. Furthermore, we propose a new memory addressing scheme based on standard memory, enabling us to avoid expensive content-addressable memory. The newly proposed weight generator replaces the original set of weights of a CNN layer. On selected CNN layers, we analyze to what extent the proposed weight generator can produce such weights whose utilization during inference leads to an acceptable accuracy drop

of CNN while memory content transferred from external memory is significantly reduced.

The rest of this chapter is organized as follows. Section 10.2 introduces the concept of symbolic regression utilizing genetic programming with memory. The idea of an on-chip weight generator as a replacement of weight reading from external memory is presented in Sect. 10.3. Both approaches are experimentally evaluated—the results are summarized in Sect. 10.4. Finally, conclusions are given in Sect. 10.5.

10.2 Symbolic Regression with GPAM

The concept of *memory* has been connected with GP in numerous ways, including individually named storage elements [8], a linear array of indexed memory elements [17], and associative memories similar to neural networks [18]. GP also automatically evolved simple abstract data structures such as stacks, queues, and lists [9].

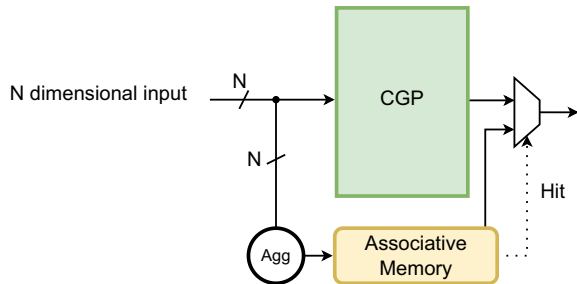
In our previous approach, GPAM, we simultaneously evolved an expression and content of associative memory for purposes of symbolic regression. In particular, GP evolves expression G and content of a small associative memory (AM), which stores a subset D_{AM} of data points from the training data set D , where $(x_i, y_i) \in D$, $|D| = n$, $|D_{AM}| = k_{AM}$, and $k_{AM} \ll n$. The resulting value y' is computed using Eq. (10.1):

$$y'(x) = \begin{cases} AM(x) & \text{if } x \text{ is in } AM \\ G(x) & \text{otherwise} \end{cases} \quad (10.1)$$

Figure 10.1 shows the overall scheme of the method. The Agg block aggregates the input data to form a key, which addresses a given value in AM . It is assumed that the condition “ x is in AM ” can be evaluated quickly. G is evolved with standard CGP [12] whose chromosome is extended with k_{AM} items referring to stored data points (x_i, y_i) . An example of encoding used in CGP is given in Fig. 10.2.

The fitness function, evaluating a given (G, AM) pair on training data set D is based on minimizing a given error metric:

Fig. 10.1 GPAM: The regression output is obtained from either the expression evolved with CGP or associative memory



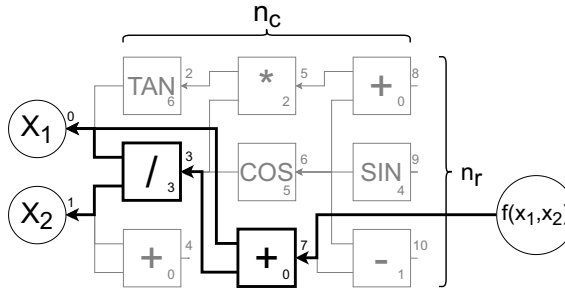


Fig. 10.2 Example of CGP representation for expression $f = x_1 + x_1/x_2$. CGP is defined using parameters: the number of primary inputs ($n_i = 2$, in our example), primary outputs ($n_o = 1$), columns ($n_c = 3$), and rows ($n_r = 3$). The function set contains 7 functions: + (encoded 0), - (1), ..., Tan (6). In the chromosome, a triplet (source₁, source₂, function) is devoted to encoding one node. The last integer specifies the output node. GPAM extends the chromosome with k_{AM} data points stored in AM . The resulting chromosome is (0, 0, 6)(0, 1, 3)(1, 1, 0)(2, 3, 2)(3, 3, 5)(0, 3, 0)(5, 6, 0)(6, 6, 4)(6, 7, 1):(7):(3, 0.1)(6, 2.1) for $k_{AM} = 2$

$$\text{Fitness}(G, AM, D) = \sum_{i=1}^n (y_i - y'(x_i))^2. \quad (10.2)$$

The standard CGP, with a common mutation operator, modifies c_m integers of the chromosome representing expression G . Another mutation operator was introduced to replace a data point (x_i, y_i) randomly picked in AM with another data point randomly taken from D . The replacement is applied with the probability p_{mutmem} .

Contrasted to [7], we will evaluate GPAM on a more comprehensive set of benchmark problems. In the experimental evaluation, we will employ test data points different from those used during the evolution. However, the random modifications of the data set (i.e., the introduced outliers) will remain preserved in the test data set. The reason for distinguishing between training and test data is to get GPAM closer to the standard GP practice. We will investigate the impact of memory sizing on the regression error when data with many outliers are present. We will deal with neither alternative aggregation functions nor utilization of constants in CGP as these techniques were already discussed in [7].

10.3 Weight Generator with GPM

The second application is motivated by the current progressive development of energy-efficient hardware accelerators of convolutional neural networks. This chapter provides motivation for our work, introduces the GPAM-based approach [7], and proposes its improvement, the GPM-based approach eliminating the need for associative memory.

10.3.1 Efficient Processing of CNNs

Hardware accelerators of CNNs are often used for inference in mobile and other embedded devices and are heavily optimized for low energy, area on a chip, and latency. Their implementations employ many processing elements performing elementary operations such as “multiply and accumulate” (MAC) in a systolic parallel architecture. CNN is usually executed layer by layer in the accelerator. Even a single layer can not usually be executed on processing elements at once due to its size. The execution has to be scheduled in many steps. Since a common CNN can have millions of weights, keeping them in local memory is impossible. Hence, they are stored in external DRAM memory. Their subsets are moved to the accelerator’s local memory buffer when scheduled by a controller. Many authors reported that memory accesses significantly contribute to the overall energy consumption of the accelerator [2, 16]. For example, while a single 8-bit multiplication only requires 0.2 pJ, reading 32 bits from local memory needs 5 pJ, and reading 32 bits from external DRAM memory consumes 640 pJ (for 45 nm technology [16]).

CNNs are highly error-resilient, so their implementation can be simplified without significantly dropping accuracy. The simplification (approximation) techniques include pruning, utilizing approximate implementations of arithmetic operations, and weight compression [1, 5]. For example, in the *weight sharing* method (also known as the *weight compression*), a group of similar weights is replaced by a single value determined using a clustering algorithm like K-means to reduce the number of weights in memory. NSGA-II was employed to find the best trade-off between the number of shared weights for each layer and the classification accuracy drop. The method introduces an error because not all weights are reconstructed perfectly. The authors reported $5\times$ compression rate for some CNN models with 32 bit weights when evaluated on the ImageNet image classification benchmark [3].

10.3.2 GPAM-Based Method

In our previous work [7], we evolved expression–memory pairs to generate weights for selected layers of CNNs. We aimed at minimizing access to external memory and, thus energy needed for one inference. The method starts with a fully trained CNN-based classifier with the classification accuracy A_c on a test data set. Let S and S_j be the original set of weights and a subset of weights ($S_j \subset S$) belonging to layer j , respectively.

The objective is to evolve a weight generator (consisting of the expression G_j and associative memory AM_j) that can sequentially approximate S_j , one weight in one step. The generated weights and incoming activations are used as the inputs to the systolic array computing output feature maps. An open question is how to generate the input sequence for CGP. In [7], we utilized the following indexes and information relevant for the j th CNN layer to create the sequence of inputs to CGP: input

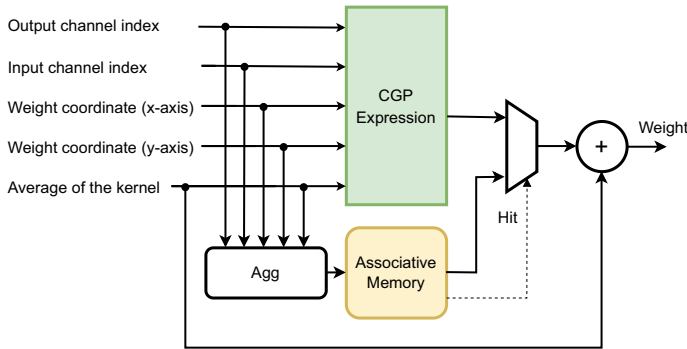


Fig. 10.3 GPAM-based weight generator for a single layer of a CNN according to [7]

channel index, output channel index, weight window coordinate w_x , weight window coordinate w_y , and the average of the weights in the kernel (w_{avg}). These items were aggregated by means of concatenation to form a 5×32 -bit input vector (a key) to associative memory. Figure 10.3 shows that the output produced by (G_j, AM_j) was then added to the average w_{avg} and interpreted as the weight at position (w_x, w_y) in the corresponding channels. The fitness of (G_j, AM_j) was obtained using Eq. (10.2); i.e., the objective was to minimize the error between the original weights and approximate weights because it is much faster than measuring the classification accuracy (for a CNN utilizing the generated weights) directly on test data.

This approach, in fact, minimizes the number of accesses to external weight memory by reducing the number of weights that have to be stored in memory. Most of the weights can be generated using the evolved expression due to weight similarity and redundancy of CNNs in general. It is assumed that the expression can be represented by a small configuration bitstream stored in external memory, which is sent into a reconfigurable weight-generating circuit when a given layer is processed by the accelerator. Note that genetic algorithms (GA) were utilized for data compression in the past [4, 14]. For example, GA determined positions of pixels serving as input to a human-created pixel predictor used in lossless image compression in [14].

In this use case, training and test data sets are not distinguished during the evolution of the expression–memory pair because GPAM always works with one particular subset of weights (S_j). The resulting expression–memory pair has to be optimized for this subset and, in fact, performs compression of this subset. Hence, the method will be evaluated as a data compression method in Sect. 10.4.2.2 and could be helpful when analyzing properties of statistical modeling of data, e.g., in the context of the minimum description length [13].

10.3.3 GPM-Based Method

The proposed GPM-based method improves the original GPAM-based approach to weight generation in the following aspects.

- (i) *The 32-bit floating-point operations used by CGP are replaced with 8-bit fixed-point operations.* The simplified operations of the CGP's function set could negatively affect the accuracy of symbolic regression. However, they significantly contribute to reducing the complexity of the potential implementation of the method in the CNN accelerator. Note that, for example, an 8-bit fixed-point multiply consumes $15.5\times$ less energy ($12.4\times$ less area) than a 32-bit fixed-point multiply, and $18.5\times$ less energy ($27.5\times$ less area) than a 32-bit floating-point multiply [16]. Another benefit is the reduced bit width of all the connections on a chip.
- (ii) *Associative memory is replaced with standard memory, and a new addressing scheme is introduced.* Associative memory is expensive because it has to store a key (obtained by the aggregation function) and the corresponding weight. Furthermore, its implementation is resource-demanding as the input key has to be compared in parallel with all the stored keys. Recall that associative memory holds a subset of h weights of the entire set of weights belonging to one layer. The proposed solution exploits the fact that we can sort the subset of weights according to the weight's order and pre-calculate the distance d_j between any two neighboring weights. The distance and weight are stored at the same address in *Memory*. Note that *Memory* is addressed with addresses $0, 1, \dots, h-1$. Figure 10.4 shows a new addressing scheme utilizing standard memory, where the data is selected based on the address (i.e., no content addressing is employed). The number *Generator* sequentially generates numbers $p = 0, 1, \dots$. The current distance d_j and weight w_j are prepared in two registers. If $p = d_j$, then the weight stored in the register represents the output of GPM, and the following steps are carried out: the address of *Memory* is increased by 1, a new distance–weight pair is read from *Memory*, stored in the registers, and the number *Generator* is reset. If $p \neq d_j$, the evolved expression generates the output weight. As the number of bits needed to represent a distance is very small, significant memory savings are obtained compared with associative memory. The proposed solution utilizes neither the weight average as the CGP's input nor the output adder.
- (iii) *Memory requirements are determined.* The memory requirements of the GPAM method can be determined as:

$$M_b = h * (k_b + w_b) + G_b, \quad (10.3)$$

where h , k_b , w_b , and G_b are the number of weights in memory, the number of bits of the key (i.e., the resulting bit width after the aggregation), the number of bits needed for one weight, and the number of bits to encode the G expression, respectively. If the GPM-based approach is considered, the key's bit width is replaced in Eq. (10.3) by the number of bits needed to store the longest distance

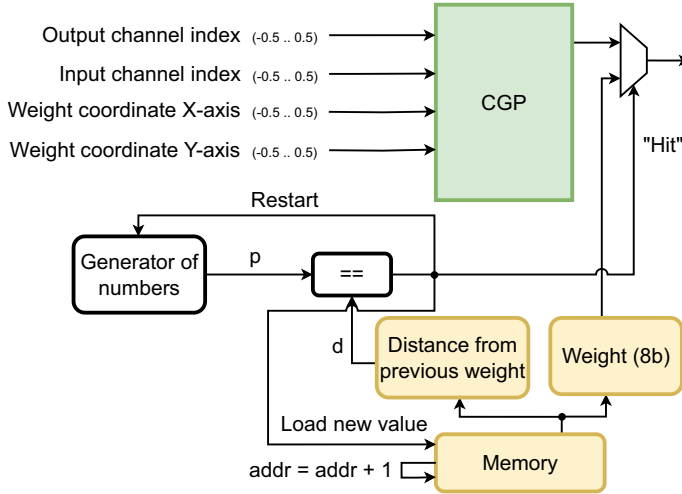


Fig. 10.4 Proposed weight generator for a single layer of a CNN. Associative memory is replaced with a common memory and a simple address-generating scheme

d between two weights. The compression rate (CR) delivered by the proposed method is

$$CR = \frac{n * w_b}{M_b}, \quad (10.4)$$

where n is the total number of weights of one layer.

10.4 Results

The experimental results are discussed in two sections—the symbolic regression on synthetic data sets and the weight generation in CNN accelerators.

10.4.1 Symbolic Regression

The goal is to show that GPAM can approximate data sets containing many random values (outliers) much better than a standard GP. We will first describe how the synthetic benchmark problems are created and then evaluate GPAM on them.

Table 10.1 Baseline functions used to generate synthetic benchmark problems for GPAM

Name	Function	Size	Interval	Step
Koza-1	$f(x) = x^4 + x^3 + x^2 + x$	40	$[-1, 1]$	0.05
Nguyen-7	$f(x) = \ln(x + 1) + \ln(x^2 + 1)$	20	$[0, 2]$	Random
Nguyen-10	$f(x_0, x_1) = 2 * \sin(x_0) * \cos(x_1)$	100	$[-1, 1]$	Random
Korns-4	$f(x_0, x_1, x_2, x_3, x_4) = -2.3 + 0.13 * \sin(x_2)$	10,000	$[-50, 50]$	Random
Keijzer-1	$f(x) = 0.3 * x * \sin(2\pi x)$	20	$[-1, 1]$	0.1
Keijzer-8	$f(x) = \sqrt{x}$	100	$[0, 100]$	1
Vladislavleva-1	$f(x_0, x_1) = \frac{e^{-(x_0-1)^2}}{1.2+(x_1-2.5)^2}$	100	$[0.3, 4]$	Random
Vladislavleva-2	$f(x) = e^{-x} x^3 (\cos(x) \sin(x)) (\cos(x) \sin^2(x) - 1)$	100	$[0.05, 10]$	0.1
Vladislavleva-5	$f(x_0, x_1, x_2) = 30 \frac{(x_0-1)(x_2-1)}{x_1^2(x_0-10)}$	300	$x_0, x_2 : [0.05, 2]$ $x_1 : [1, 2]$	Random

10.4.1.1 Creating Synthetic Benchmark Problems

We consider nine well-known benchmark functions that are typically utilized to evaluate the GP-based symbolic regression methods [11]. Table 10.1 defines these benchmark functions, the intervals used for sampling, and the number of sampled points. We created training sets for GPAM by sampling these functions. In each data set, we replaced some percentage of data points (controlled by the parameter τ) with randomly generated values from interval $[-10, 10]$ to generate a benchmark problem with “outliers”. Then, we created test sets by (1) reusing the randomly generated data points from the training sets and (2) adding new data points by sampling the original functions again. It holds that training and test sets have the same size and contain the same randomly generated data points (“outliers”). It has to be emphasized that GPAM does not know where the randomly generated data points are located.

10.4.1.2 Setup

Based on some trial runs and with respect to results of [7], the CGP parameters are defined as summarized in Table 10.2. CGP employs different function sets for different benchmarks; see Table 10.3. According to [12], CGP employs the $(1 + \lambda)$ search strategy, where λ offspring individuals are generated from one parent. The fitness function is as defined by Eq. (10.2).

Table 10.2 Parameters of CGP when used in GPAM

Parameter	Value
n_i	Function dimension
n_o	1
$n_c \times n_r$ (columns $\times$ rows)	20×2
L-back	Maximum
Search method	$1 + \lambda$, $\lambda = 4$
Generations	5000
c_m	2 integers per chromosome
p_{mutmem}	0.2 (the mutation probability for memory)
Float constant probability	0.05 (0.10 if memory size = 0)
AM constant probability	0.05 (0.00 if memory size = 0)

Table 10.3 Function sets used by CGP for different benchmarks

Function name	Function set
Koza, Nguyen	$+, -, *, \%, \sin, \cos, e^x, \log(x)$
Korns	$+, -, *, \%, \sin, \cos, e^x, \log(x), x^2, x^3, \text{sqrt}, \tan, \tanh$
Keijzer	$+, *, \frac{1}{x}, -x, \text{sqrt}$
Vladislavleva-5	$+, -, *, \%, x^2$
Vladislavleva-1	$+, -, *, \%, x^2, e^x, e^{-x}$
Vladislavleva-2	$+, -, *, \%, x^2, e^x, e^{-x}, \sin, \cos$

10.4.1.3 Example Result

To illustrate the behavior of the proposed approach, let us first analyze an example result evolved by GPAM—the approximation of the Nguyen-7 benchmark (with $\tau = 50\%$, i.e., 10 modified values) when GPAM has no memory ($k_{AM} = 0\%$) and when it can use memory whose size is 60% of original data points. For $k_{AM} = 0\%$, Fig. 10.5 shows that CGP evolved a very complex expression (Eq. (10.5)) trying to fit as many data points as possible.

$$\begin{aligned}
 f(x) = & \frac{1}{x} (x(-2x \cos(x) - \ln \left(\frac{|(2x \cos(x) - \cos(2x)) \ln(2|x \cos(x)|)|}{x \cos(x)} \right) |) \\
 & + \ln(|\cos(2x)|) - \sin \left(\frac{\ln(|\cos(2x)|)}{x} \right) + \cos(2x)) \\
 & - (e^{\cos(x)} * x - \ln(|\cos(2x)|)) \sin(\ln(|\cos(2x)|)x))
 \end{aligned} \tag{10.5}$$

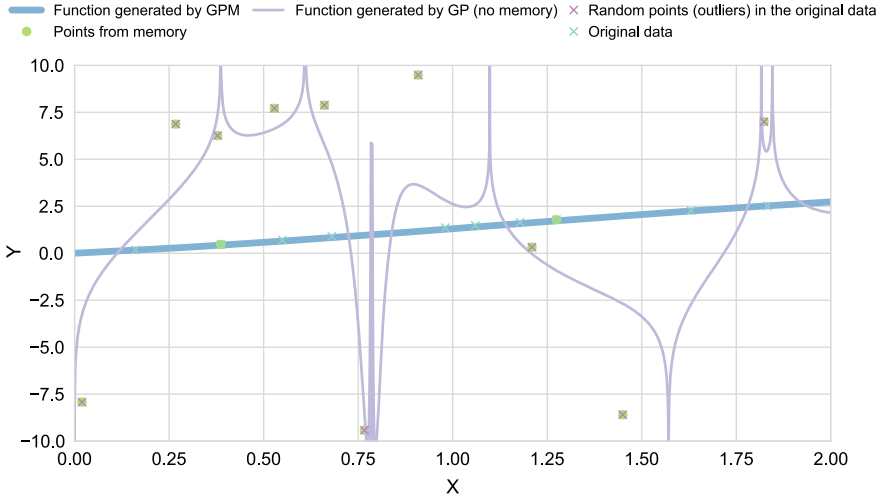


Fig. 10.5 The Nguyen-7 benchmark problem: original data points (blue $\times$) and randomly generated points (purple $\times$). One of the evolved solutions with $k_{AM} = 60\%$ is depicted in blue; green points (circles) are taken from AM . One of the evolved solutions with $k_{AM} = 0\%$ is depicted in purple

However, in the case that associative memory is available, the evolved expression (Eq. 10.6) is simple:

$$f(x) = \ln(|e^x + x^3|) \quad (10.6)$$

and most “outliers” are correctly stored in memory (see the green points in Fig. 10.5).

10.4.1.4 The Role of Memory Sizing

We performed 30 independent runs of GPAM with various settings of k_{AM} and τ for all benchmark data sets. Figure 10.6 shows in detail on the Keijzer-8 benchmark that the fitness values (i.e., the approximation error on training as well as test data) decrease with lowering the number of randomly generated data points (τ) in the data set and adding more memory (k_{AM}). If $\tau = 0$ and $k_{AM} = 0$, then GPAM becomes a standard CGP, which almost always delivers a solution showing a close to zero error. Note that the fitness values are normalized with respect to the number of data points in all figures showing box plots in this paper.

Figure 10.7 shows the same plot for the remaining benchmark problems. Contrasted to [7], all these results are presented for test sets, i.e., the data unseen during the evolution (except the randomly generated data points). The exact obtained values are less important because they can slightly differ when another GP version is utilized. However, the following trend is significant: GPAM almost always provides better solutions for less contaminated data, and when the size of AM is increasing.

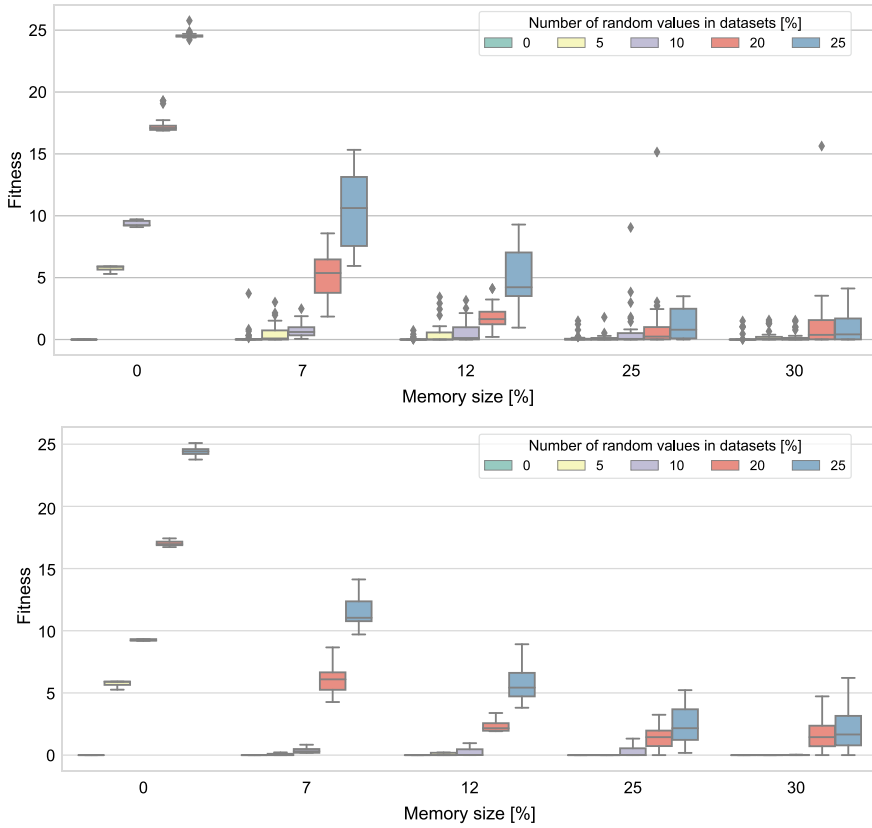


Fig. 10.6 Fitness values obtained by GPAM on Keijzer-8 benchmark problem for various associative memory sizes (k_{AM}) and numbers of randomly modified data points in data sets (τ). Box plots are constructed using training data (top) and test data (bottom); for the latter case, outliers in the box plots are omitted

We can conclude that GPAM can evolve a suitable expression and choose appropriate data points to be stored in associative memory to approximate non-trivial data sets.

The experiments were performed on the Intel Xeon CPUs E5-2630 v4 running at 2.20 GHz. For the most demanding setup, the median execution time is 280 s on a single core.

10.4.2 Weight Generation

The weight generation method based on GPM is evaluated on several layers of selected CNNs trained as image classifiers. We start with a simple CNN classifying the MNIST data set, where GPAM and GPM are compared. Then, the same approach is adopted for more complex CNNs.

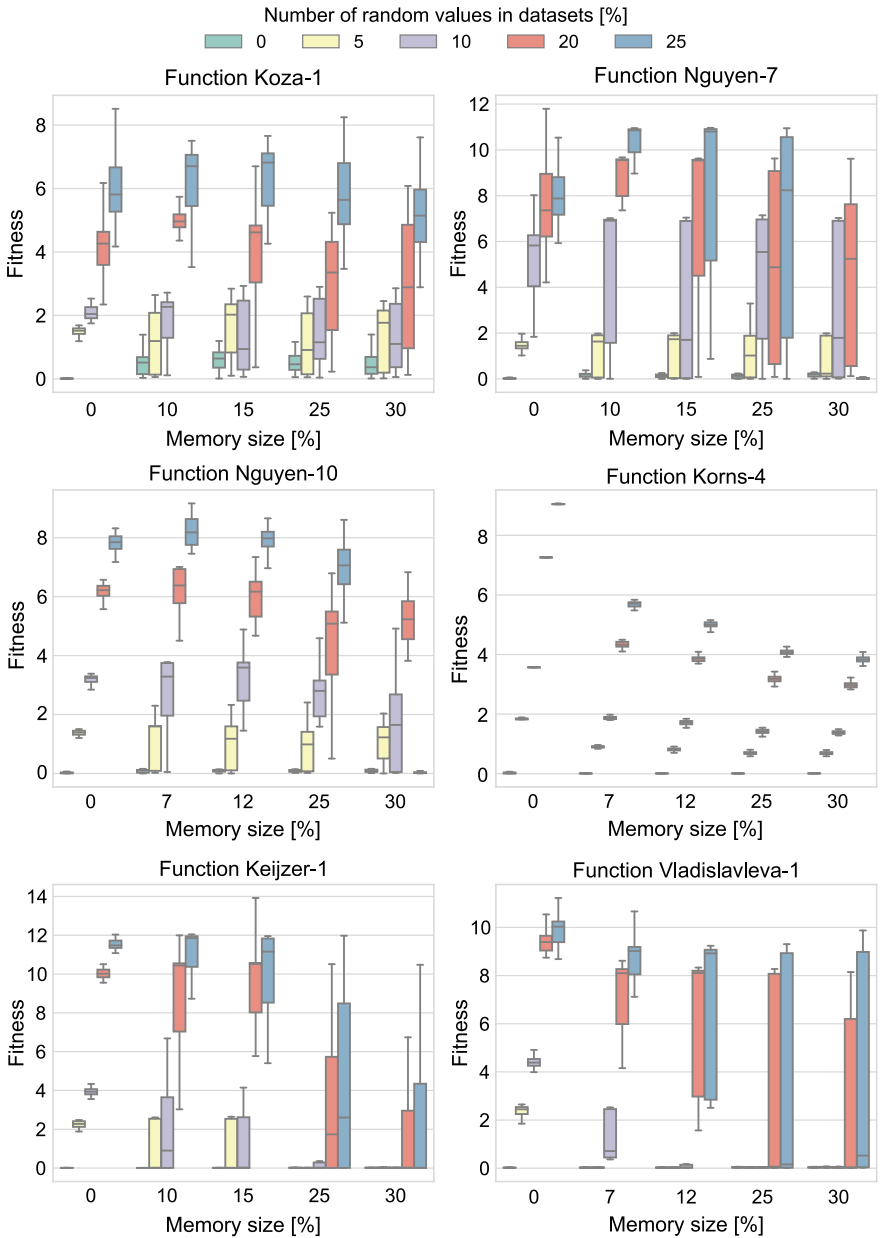


Fig. 10.7 Fitness values obtained by GPAM on eight benchmark problems for various associative memory sizes (k_{AM}) and numbers of randomly modified data points in data sets (τ)

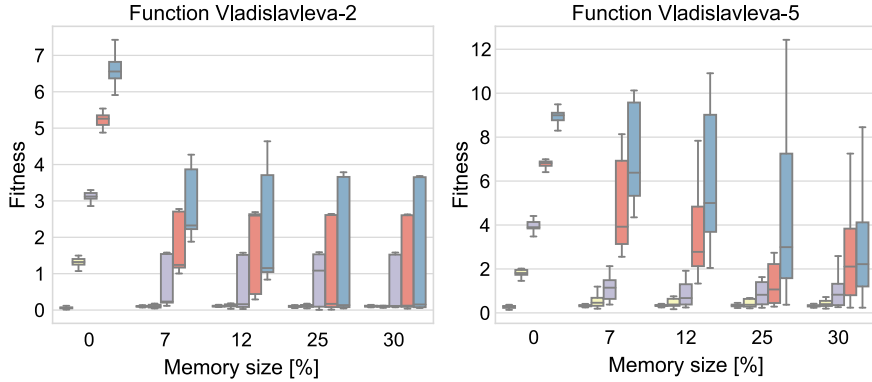


Fig. 10.7 (continued)

10.4.2.1 Simple CNN 1

The first CNN, denoted Simple CNN 1, consists of six layers; see Fig. 10.8. This CNN was trained using PyTorch to classify the MNIST data set (a digit classification task from a 28×28 -pixel image [10]) and achieved 97.4% accuracy on test images. When the 8-bit precision is applied, the classification accuracy is 97.3%. We created two data sets; S1 contains 250 weights of the first convolutional layer, and S2 contains 5000 weights of the second convolutional layer; see their values plotted in Fig. 10.9.

Table 10.4 summarizes the setup of CGP parameters for the evolutionary design of weight generators. The setup is based on the results of the previous study [7] and a few additional test runs. GPAM utilizes floating-point data representation, operates with the following function set $\Gamma_{32b} = \{+, -, *, \%, \sin(x), \cos(x), e^x, \log(x), x^2, x^3, \sqrt{x}, \tan(x), \tanh(x)\}$, and constants can randomly be generated. The eight-bit function set used by GPM is $\Gamma_{8b} = \{x, \text{not}(x), \text{rshift1}, \text{rshift2}, \text{lshift1}, \text{lshift2}, \text{or}, \text{and}, \text{xor}, +, -, *, \text{const00}, \text{constFF}\}$. It includes two constants, 00 and FF. A weight is represented on 8 bits so that 1 bit is reserved for the sign, and the remaining (7) bits represent the interval $[0; 0.5]$.

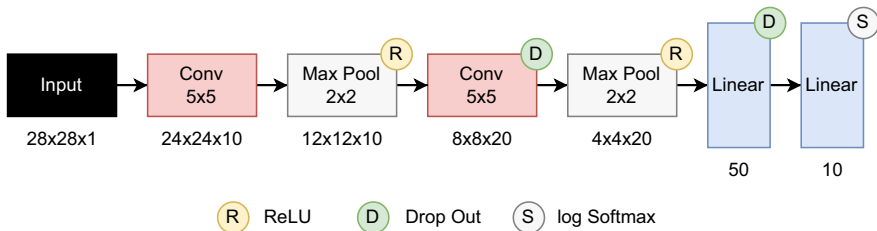


Fig. 10.8 CNN used as a benchmark problem for GPAM

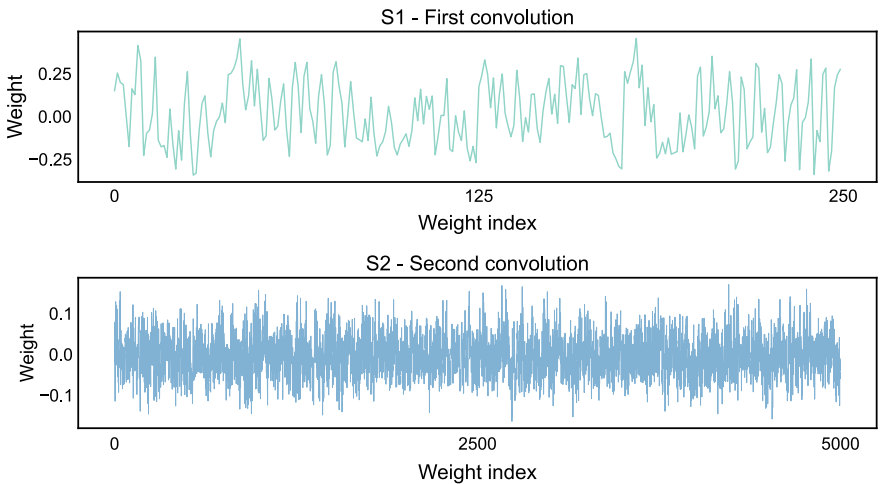


Fig. 10.9 Data sets created using the weights belonging to the first (S1) and second (S2) convolutional layer of Simple CNN 1

Table 10.4 Parameters of CGP when evolving the weight generator

Parameter	Value
n_i	4
n_o	1
$n_c \times n_r$ (columns $\times$ rows)	20×10
L-back	Maximum
Search method	$1 + \lambda$, $\lambda = 4$
Generations	5000
c_m	2 (the integers to mutate in CGP)
p_{mutmem}	0.2 (mutation probability for memory)
Probability of creating a constant	0.05 (GPAM)

Figure 10.10 shows box plots with the fitness values obtained for S1 (left) and S2 (right) from 30 independent runs of GPAM (green) and GPM (yellow) utilizing various sizing of memory. We observe the same trend as in the previous experiments; the classification accuracy is improved if a bigger memory is available. If no memory is considered, the classification accuracy is not competitive at all. For memory containing 10% of the weights (selected by the method automatically), the classification accuracy is close to the baseline values. The best accuracy obtained by GPM (the 8-bit setup) is close to the 32-bit GPAM. The statistical evaluation expressed by the box plots reveals that GPAM consistently provides higher medians of accuracy for S2. Interestingly, GPM leads to higher medians for S1.

Equation (10.7) gives one of the best-evolved 8-bit expressions for S1. Surprisingly, it uses only two inputs (x_1 and x_2). For S2, GPM delivered an expression

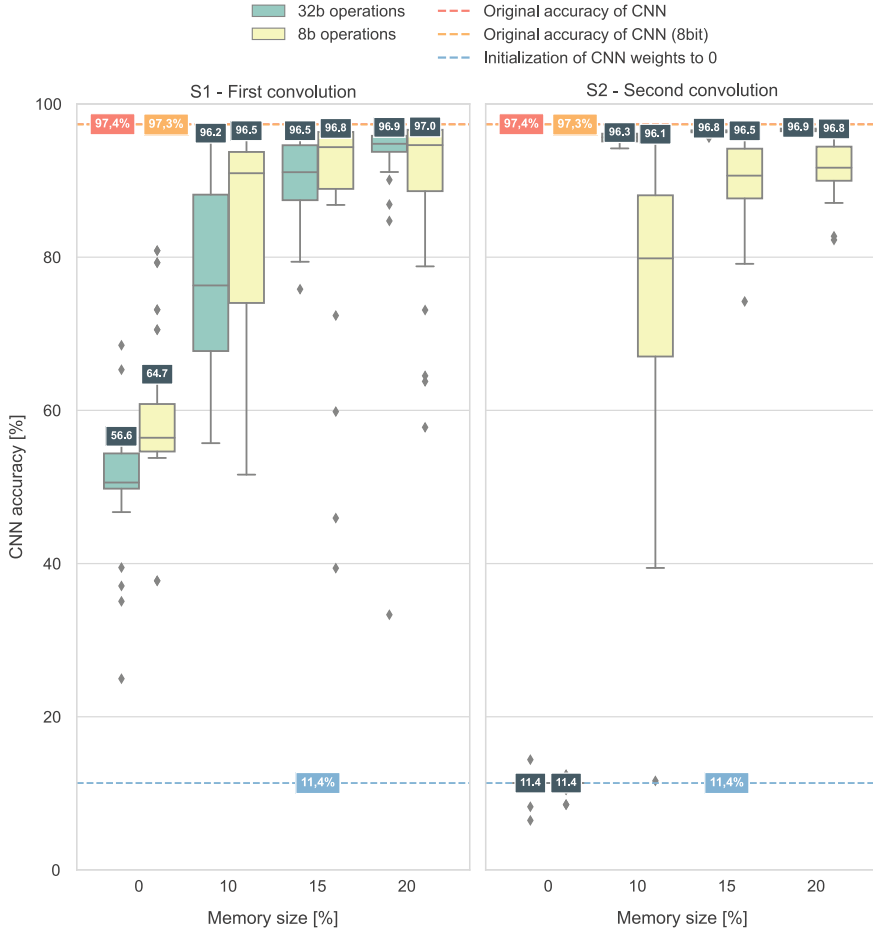


Fig. 10.10 Accuracy (on the MNIST test data) obtained from 30 runs of GPAM (green) and GPM (yellow) utilizing various sizing of memory on S1 and S2 data sets. The numbers in rectangles represent the best-obtained values under a given setup

(Eq. (10.8)), which is a constant. It seems that for simple CNNs and classification problems, it is enough to use a simple expression, even a constant expression, and 10%-20% weights to approximate the original weight set with a small accuracy drop.

$$G_{S1}(x_0, x_1, x_2, x_3) = ((FF \gg 2) * (((((x_1 \text{ xor } x_2) - 00) \text{ or } ((x_1 \text{ xor } x_2) - 00)) \text{ or } (x_1 \text{ xor } x_2)) \text{ and } ((\sim (x_2 \text{ and } (x_1 \text{ xor } x_2))) \text{ or } (00 \ll 2)))))) \quad (10.7)$$

$$G_{S2}(x_0, x_1, x_2, x_3) = \sim (x_2 \text{ xor } x_2) = -0.00390625. \quad (10.8)$$

Table 10.5 The number of bits obtained with several methods to represent the weights of convolutional layers captured by data sets S1 and S2

Method/data set	S1 ($n = 250$) (bit)	S2 ($n = 5\,000$) (bit)
Original size (32 bit)	8 000	160 000
Original size (8 bit)	2 000	40 000
Huffman coding	1 737	28 183
Huffman coding (incl. the table)	3 867	29 355
ZIP	3 344	39 464
Theoretical optimum (entropy)	1 732	28 067
GPM/GPAM, $k_{AM} = 10\%$ (Acc. drop 0.8%)	636/3698	–
GPM/GPAM, $k_{AM} = 10\%$ (Acc. drop 1.0%)	448/3484	–
GPM/GPAM, $k_{AM} = 10\%$ (Acc. drop 1.2%)	–	6765/68044

10.4.2.2 Memory Requirements

A common memory subsystem of a CNN accelerator would require external memory capacity $w_b \times n$ bits for n weights represented on w_b bits. Table 10.5 summarizes the number of bits obtained with several compression methods to represent the weights of convolutional layers captured by data sets S1 and S2. We included the lossless Huffman code with and without the symbol table, the ZIP compression algorithm, and the theoretical optimum computed using entropy (note that the frequency of occurrence of the weight values is known for S1 and S2). Then, the best-evolved solutions obtained by GPM and GPAM are reported. The solutions evolved by GPM led to the smallest memory requirements; however, an accuracy drop is introduced as the weights are not reconstructed perfectly. For GPM and GPAM, the number of bits was obtained using Eq. (10.3), which considers the number of weights in local memory, their bit width, the maximum distance between two weights (which is typically encoded on 6 or 7 bits), and the size of CGP chromosome needed to encode the evolved expression (100–300 bits).

Regarding the computational requirements, for the most demanding setup (30% weights in memory), the median single-core execution time of GPM is 850 s on the Intel Xeon CPU E5-2630 v4 running at 2.20 GHz.

10.4.2.3 Other CNNs

We repeated the experiments from Sect. 10.4.2.1 on selected convolutional layers of some CNNs trained to classify other image data sets. Particularly, we dealt with another simple CNN trained on Fashion MNIST (Simple CNN 2 with an accuracy of 0.91), MobileNetV2 [15] on CIFAR-10, and ResNet-34 [6] on Imagenette. The

Table 10.6 The basic setup of GPM and the obtained trade-offs between the accuracy drop and memory size (k_{AM})

Weight set	CNN	Data set	Layer	#Weights	Bits	Acc. drop (%)	k_{AM} (%)
S1	Simple CNN 1	MNIST	1	250	8	0.8	10
S1	Simple CNN 1	MNIST	1	250	8	0.3	20
S2	Simple CNN 1	MNIST	2	5000	8	1.2	10
S2	Simple CNN 1	MNIST	2	5000	32	0.5	20
S3	Simple CNN 2	FMNIST	1	400	32	7.4	20
S4	Simple CNN 2	FMNIST	2	12800	32	11.3	15
S5	MobileNetV2	CIFAR-10	1	864	32	6.7	15
S6	MobileNetV2	CIFAR-10	3	288	32	1.3	15
S7	MobileNetV2	CIFAR-10	5	864	32	8.9	15
S8	ResNet-34	Imagenette	1	9408	32	37.5	30
S9	ResNet-34	Imagenette	2	36864	32	3.4	30
S10	ResNet-34	Imagenette	2 & 3	73728	32	19.8	10

composed sets of weights are denoted S3 - S10. Table 10.6 summarizes the basic setup and the obtained trade-offs between the accuracy drop and memory size needed by GPM. One can observe that the weights of some layers can be compressed so that only a small accuracy drop is present; however, if CNN is more complex, the accuracy drop becomes unacceptable.

10.5 Conclusions

We proposed GPM as an extension of our previous method GPAM, both dealing with the automated design of programs that can utilize small memory to memorize some important information from a training data set. We showed that the method can generalize on common data sets (infected with randomly generated values), which has not been demonstrated in [7].

We evolved expression-memory pairs that can serve as weight generators and thus approximate the weights associated with convolutional layers of selected CNNs. For example, if memory contains 10% of the original weights, the weight generator evolved for a convolutional layer can approximate the original weights such that the CNN utilizing the generated weights shows less than a 1% drop in the classification accuracy on the MNIST data set. The memory requirements are reduced $3.1\times$ or $12.6\times$ for 8-bit or 32-bit weights, respectively. The same approach was adopted for more complex CNNs and challenging image data sets. However, a detailed evaluation of the implementation on a particular hardware accelerator or its simulator has to be performed to obtain all important hardware characteristics, such as latency and energy of inference.

The proposed method can be extended in several directions, for example, by considering the importance of particular weights for the CNN accuracy in the fitness function, employing different GP variants, or creating the key for associative memory in alternative ways. Its utilization in other applications (such as logic circuits for machine learning or image filtering) is our future task.

References

1. Armeniakos, G., Zervakis, G., Soudris, D., Henkel, J.: Hardware approximate techniques for deep neural network accelerators: a survey. *ACM Comput. Surv.* **55**(4), 83:1–83:36 (2023)
2. Capra, M., Bussolino, B., Marchisio, A., Shafique, M., Masera, G., Martina, M.: An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks. *Future Internet* **12**(7), 113 (2020)
3. Dupuis, E., Novo, D., O'Connor, I., Bosio, A.: A heuristic exploration of retraining-free weight-sharing for CNN compression. In: 27th Asia and South Pacific Design Automation Conference, ASP-DAC, pp. 134–139. IEEE (2022)
4. Grasmann, U., Miikkulainen, R.: Effective image compression using evolved wavelets. In: Beyer, H., O'Reilly, U. (eds.) *Genetic and Evolutionary Computation Conference*, pp. 1961–1968. ACM (2005)
5. Han, S., Mao, H., Dally, W.J.: Deep compression: compressing deep neural network with pruning, trained quantization and Huffman coding. In: 4th International Conference on Learning Representations, ICLR (2016)
6. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016)
7. Juza, T., Sekanina, L.: GPAM: genetic programming with associative memory. In: *Genetic Programming—26th European Conference, EuroGP 2023*. LNCS, vol. 13986, pp. 68–83. Springer (2023)
8. Koza, J.R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge and MIT Press, London, Bradford Book (1992)
9. Langdon, W.B.: *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!* Springer (1998)
10. LeCun, Y., Cortes, C., Burges, C.: MNIST handwritten digit database. ATT Labs [Online], 2 (2010). <http://yann.lecun.com/exdb/mnist>
11. McDermott, J., White, D., Luke, S., Manzoni, L., Castelli, M., Vanneschi, L., Jaśkowski, W., Krawiec, K., Harper, R., De Jong, K., O'Reilly, U.M.: Genetic programming needs better benchmarks. In: *Proceedings of the 14th International Conference on Genetic and Evolutionary Computation*, pp. 791–798. ACM (2012)
12. Miller, J.F.: *Cartesian Genetic Programming*. Springer, Berlin Heidelberg (2011)
13. Rissanen, J.: *Information and Complexity in Statistical Modeling*. Springer, New York, NY (2007)
14. Sakanashi, H., Iwata, M., Higuchi, T.: EHW applied to image data compression. In: Higuchi, T., Liu, Y., Yao, X. (eds.) *Evolvable Hardware*, pp. 19–40. Springer (2006)
15. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.: Mobilenetv2: Inverted residuals and linear bottlenecks. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 4510–4520. IEEE Computer Society (2018)
16. Sze, V., Chen, Y., Yang, T., Emer, J.S.: *Efficient Processing of Deep Neural Networks*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers (2020)
17. Teller, A.: The evolution of mental models. In: Kinnear, Jr. K.E. (ed.) *Advances in Genetic Programming*, pp. 199–219. MIT Press (1994)

18. Villegas-Cortéz, J., Olague, G., Avilés-Cruz, C., Sossa, H., Ferreyra, A.: Automatic synthesis of associative memories through genetic programming: a first co-evolutionary approach. In: Applications of Evolutionary Computation. LNCS, vol. 6024, pp. 344–351. Springer (2010)
19. Volder, J.E.: The CORDIC trigonometric computing technique. IRE Trans. Electron. Comput. **EC-8**, 330–334 (1959)

Chapter 11

Ratcheted Random Search for Self-programming Boolean Networks



Lee Spector

Abstract Random Boolean networks exhibit a variety of properties that are characteristic of natural adaptive systems. This suggests that Boolean networks may provide a useful substrate for artificial systems that adapt to solve specified problems. In this chapter, I describe an investigation of the capacity of Boolean networks to represent solutions to computational problems and the efficacy of simple stochastic algorithms for finding problem-solving Boolean networks. Because prior work noted that the properties of Boolean networks depend on the set of permitted gate types, I consider here networks constructed only of programmable gates that can be configured to act as any 2-input Boolean gate. Networks of these gates are self-programming because the function of each gate is determined dynamically from its configuration inputs. I consider search algorithms that find self-programming networks through processes of random rewiring, random gate additions, random gate deletions, and a ratchet mechanism that only permits moves in the search space that maintain or increase training case coverage. I present networks found by this algorithm and discuss the implications of these results for future work.

11.1 Introduction

Starting in 1969, Stuart Kauffman and subsequently others have explored the properties of randomly wired networks of Boolean gates. Much of this work has been motivated by an interest in illustrating features of genetic regulatory networks and other complex biological systems [3, 7–9, 21]. Among the features of random Boolean networks that have been the focus of this work are the spontaneous emergence of order, attractors with modest cycle lengths in spite of the enormous state spaces in which they occur, dynamics that can be tuned to the “edge of chaos” to facilitate adaptation, and resilience to perturbation.

One suggestion raised by this research, and explored to some extent in prior work, is that Boolean networks may be useful as targets of adaptive processes that aim to

L. Spector (✉)
Amherst College, Amherst, Massachusetts, USA
e-mail: lspector@amherst.edu

produce executable structures that solve specified problems. For example, Kauffman considered the effects of wire-moving mutations and the prospects for adaptive walks through the space of wirings to find attractors with particular patterns of activation for specified subsets of gates [8]. Lemke et al. presented a more thorough numerical study of adaptation in populations of Boolean networks evolved using genetic algorithms [10], again in the context of finding networks with attractors matching specified patterns. East and Rowe considered the use of Kauffman's Boolean networks in the developmental mechanisms of a system that evolved problem-solving neural networks [2]. Gershenson described methods for guiding the self-organization of random Boolean networks to different regimes of network dynamics [4]. Although this and other related work have revealed and to some extent exploited interesting features of Boolean networks, it does not appear that work on these approaches has yet provided a robust technology for finding Boolean networks that compute specified functions.

Separately, work on evolvable hardware and Cartesian genetic programming has produced methods that can find Boolean circuits that compute specified functions [5, 12, 13, 16, 18, 19]. Most of this work has focused on finding combinatorial circuits, which cannot have cycles and hence cannot exhibit the attractor dynamics that has driven interest in Boolean networks. Prior work on finding sequential circuits, which may have cycles, has focused on the evolution of circuits with highly constrained architectures for sequential processing [1, 15]. Little work appears to have been done so far on the adaptation of unconstrained Boolean networks to compute specified functions that map Boolean inputs to Boolean outputs.

The present study aims to take initial steps in the filling of this gap, by exploring the search for unconstrained Boolean networks that compute functions of interest. I assume a problem environment in which an input consists of some fixed number of Boolean values, and in which an output consists of some possibly different fixed number of Boolean values.

Because the goal at this stage is to uncover basic principles of search for networks that implement target functions, choices of representations and algorithms have been made with primary attention to considerations of simplicity, generality, and uniformity. One choice reflecting this priority is that I consider here only networks made of one kind of gate, a "programmable" gate that can act as any 2-input Boolean gate. This is in contrast to prior work that considered heterogeneous networks composed of gates chosen from a specified set such as {AND, OR, NOT}. Another choice reflecting my focus on basic principles is that I consider only search methods that sequentially modify a single network until it meets the target specification. Relative to approaches based on more sophisticated methods such as genetic algorithms, this avoids complexities such as those introduced by interactions within populations. It is possible that future work will determine that the most effective search algorithms take advantage of such complexities, but because my current goal is to understand fundamental features of the search space I aim for simplicity, generality, and uniformity when possible here.

In the following sections I describe the programmable gate out of which the networks that I will consider are constructed, preliminary observations about the

behavior of self-programming Boolean networks, and the ways in which networks of these gates can be used to compute functions from inputs to outputs. I then describe the search method that I use to find networks that satisfy particular input/output specifications, and I present results of preliminary experiments using this search algorithm. I conclude with some general observations and suggestions for future work.

11.2 Programmable Gates

Prior work on random Boolean networks showed that a variety of network properties depend in critical ways on the subset of possible gate types that one allows to be used in the networks. In the interest of simplicity, generality, and uniformity, I have chosen here to use only a single type of gate that can be configured to act as any 2-input Boolean gate.

The function of a 2-input Boolean gate can be specified using a truth table, the right-most column of which indicates the output for each possible combination of input values. For example, the right-most column of the truth table shown in Table 11.1 specifies the behavior of an AND gate.

A programmable 2-input gate takes four additional inputs, the values of which specify the right-most column of the truth table for the Boolean function that the gate will implement. For example, if the values of the four additional inputs are `false`, `false`, `false`, and `true`, then the gate will act as an AND gate of the original two inputs. As will be useful for execution traces presented below, the right-most column of the truth table can be notated with a single hexadecimal digit by first considering it to be a binary number with the most significant bit on top. Using this convention, the AND gate is notated as 1; the hexadecimal notation for all 2-input Boolean gates is shown in Table 11.2.

The truth table for the programmable gate is shown in Table 11.3.

The programmable gate is equivalent to the well known 4-to-1 multiplexer [20], which also takes a total of six inputs and is generally discussed as using two of

Table 11.1 The truth table for an AND gate, shown using 0 for `false` and 1 for `true`

A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

Table 11.2 Hexadecimal notation for all 2-input Boolean gates. The first four columns show the right-most column of the gate’s truth table, using 0 for `false` and 1 for `true`. The column header “FF” indicates that the value provides the output of the gate when the A and B inputs are both `false`, while the “FT” header means that the value provides the output when the A input is `false` and the B input is `true`, and so on. The “Hex” column shows the hexadecimal notation for the gate with that truth table. The “Gate” column provides a common description for that gate, assuming that the first input is A and the second input is B

FF	FT	TF	TT	Hex	Gate
0	0	0	0	0	FALSE
0	0	0	1	1	AND
0	0	1	0	2	$A \wedge \neg B$
0	0	1	1	3	A
0	1	0	0	4	$B \wedge \neg A$
0	1	0	1	5	B
0	1	1	0	6	XOR
0	1	1	1	7	OR
1	0	0	0	8	NOR
1	0	0	1	9	=
1	0	1	0	A	$\neg B$
1	0	1	1	B	$B \implies A$
1	1	0	0	C	$\neg A$
1	1	0	1	D	$A \implies B$
1	1	1	0	E	NAND
1	1	1	1	F	TRUE

the inputs to specify which of the other four inputs will appear at the output.¹ This turns out to be equivalent to using the latter four inputs to specify the truth table of the Boolean function applied to the original two inputs. Figure 11.1 shows the standard graphical representation for a 4-to-1 multiplexer, while Fig. 11.2 shows my preferred graphical representation for the programmable gate. The gates are identical in function, but I prefer the “programmable” terminology and the representation in Fig. 11.2 because they remind us that this gate can act as any 2-input Boolean gate, with the choice of gate being specified by additional inputs.

¹ Thanks to Bill Tozier for pointing this out. Others have studied the evolution of circuits composed of multiplexers (e.g. [11]), but not, so far as I know, for the production of unconstrained Boolean networks.

Table 11.3 The truth table for the programmable 2-input Boolean gate, which is equivalent to that for a 4-to-1 multiplexer, shown using 0 for *false* and 1 for *true*. The column header “FF” indicates that the value provides the output of the gate when the A and B inputs are both *false*, while the “FT” header means that the value provides the output when the A input is *false* and the B input is *true*, and so on

FF	FT	TF	TT	A	B	OUT
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	0	1	0	1	0
0	0	0	1	1	0	0
0	0	0	1	1	1	1
0	0	1	0	0	0	0
0	0	1	0	0	1	0
0	0	1	0	1	0	1
0	0	1	0	1	1	0
0	0	1	1	0	0	0
0	0	1	1	0	1	0
0	0	1	1	1	0	1
0	0	1	1	1	1	1
0	1	0	0	0	0	0
0	1	0	0	0	1	1
0	1	0	0	1	0	0
0	1	0	0	1	1	0
0	1	0	1	0	0	0
0	1	0	1	0	1	1
0	1	0	1	1	0	0
0	1	0	1	1	1	1
0	1	1	0	0	0	0
0	1	1	0	0	1	1
0	1	1	0	1	0	1
0	1	1	0	1	1	0
0	1	1	1	0	0	0
0	1	1	1	0	1	1
0	1	1	1	1	0	1
0	1	1	1	1	1	1
1	0	0	0	0	0	1
1	0	0	0	0	1	0
1	0	0	0	1	0	0
1	0	0	0	1	1	0
1	0	0	1	0	0	1

(continued)

Table 11.3 (continued)

FF	FT	TF	TT	A	B	OUT
1	0	0	1	0	1	0
1	0	0	1	1	0	0
1	0	0	1	1	1	1
1	0	1	0	0	0	1
1	0	1	0	0	1	0
1	0	1	0	1	0	1
1	0	1	0	1	1	0
1	0	1	1	0	0	1
1	0	1	1	0	1	0
1	0	1	1	1	0	1
1	0	1	1	1	1	1
1	1	0	0	0	0	1
1	1	0	0	0	1	1
1	1	0	0	1	0	0
1	1	0	0	1	1	0
1	1	0	1	0	0	1
1	1	0	1	0	1	1
1	1	0	1	1	0	0
1	1	0	1	1	1	1
1	1	1	0	0	0	1
1	1	1	0	0	1	1
1	1	1	0	1	0	1
1	1	1	0	1	1	0
1	1	1	1	0	0	1
1	1	1	1	0	1	1
1	1	1	1	1	0	1
1	1	1	1	1	1	1

Fig. 11.1 A 4-to-1 multiplexer with inputs FF, FT, TF, and TT and switching bits A and B

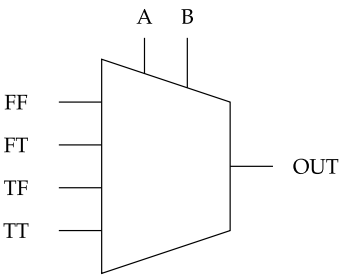
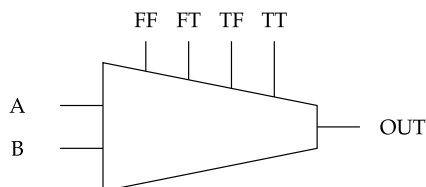


Fig. 11.2 A programmable 2-input Boolean gate with inputs A and B and truth table defined by FF, FT, TF, and TT



11.3 Self-programming Networks

A random self-programming Boolean network of a specified size can be created by creating the specified number of programmable gates and then wiring each input of each gate to the output of a randomly chosen gate. Although various network update schemes might be considered, I consider here only synchronous updates in which all gates simultaneously determine their states for step $s + 1$ by using the values of the state at step s as inputs. Note that this means that the execution of a network is entirely deterministic. While I am looking here at randomly wired networks, and I will later consider random processes for finding networks that behave in certain ways, there is nothing random in the execution of a network once it has been created.

One set of questions that immediately arises concerns the dynamics of random self-programming Boolean networks. What will they generally be like? Although studies have been conducted of the dynamics of networks of various types of Boolean gates, I am not aware of studies of networks of programmable gates in particular.

As a preliminary step in the study of these networks, I generated 10,000 networks of each size from 1 to 100 gates and ran each from a random initial state until a state was repeated, meaning that the network had entered a cycle. I then reported the length of the cycle. Figure 11.3 shows the mean cycle length for each number of gates, while Fig. 11.4 shows the maximum observed cycle length for each number of gates.

From results reported by Kauffman, one might expect the dynamics of networks of 6-input gates with true/false-balanced outputs to be quite chaotic, since he reports a tendency for chaotic dynamics in networks in which gates have more than two inputs ($K > 2$, in his terminology) and also in which gates have balanced outputs ($P = 0.5$ in his terminology). Nonetheless, the behavior of the specific gate used here appears to be more manageable, with mean cycle lengths growing modestly with the number of gates. Even for networks of 100 gates, for which the size of the state space is $2^{100} > 10^{30}$, one can expect convergence to an attractor cycle—a sequence of network states that henceforth repeats forever—of fewer than six steps. Maximum cycle lengths are subject to outliers in the thousands, but even so their growth is modest in comparison to state space size. For all network sizes the median cycle length was 1.

Although the dynamics seen here are encouraging, they reflect the behavior of randomly initialized networks. When we use networks to compute functions from

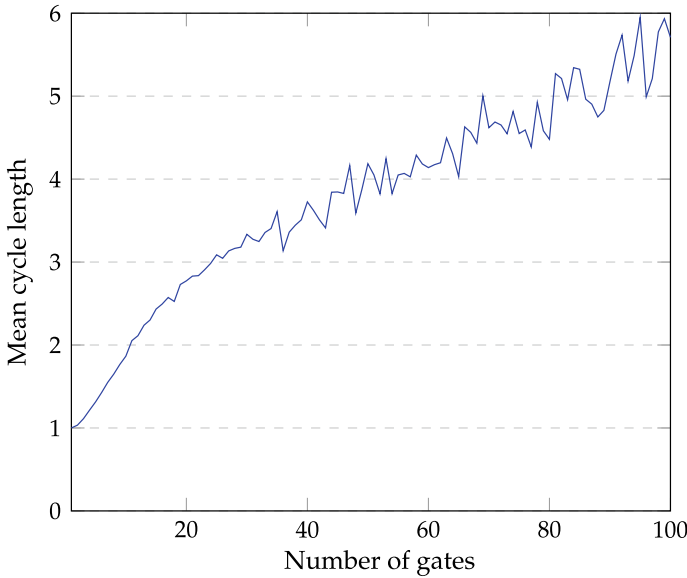


Fig. 11.3 Mean cycle lengths for random networks of programmable gates. For each plotted point 10,000 random networks of the specified size were created, initialized randomly, and run until a state was repeated

inputs to outputs, initial states will presumably be determined by inputs, and may be constrained. This may change the range of dynamics that we can expect to observe.

In order to use a self-programming Boolean network as a function that maps inputs to outputs the model is augmented with additional input sources and a convention for extracting outputs. Specifically, when working on a problem involving i Boolean inputs, $i + 2$ sources are added for wires to gate inputs beyond those provided by the outputs of the gates in the network. i of the added sources are set to provide the values of the corresponding inputs, held constant throughout the process of determining the network's outputs for those inputs. The other two added sources provide constant values of `true` and `false`. Note that this means that it is possible to wire a self-programming Boolean network to act as a network of standard 2-input gates, by wiring all of the programming inputs of all of the gates to constants. On the other hand, if the programming inputs of gates are wired to the outputs of other gates then more complex behaviors may result.

Outputs are taken from pre-specified gates in the network, so a network for a problem involving u output bits must include at least u gates that are pre-specified to serve as outputs. I refer to gates that are not designated as outputs as “hidden” gates, in analogy to the use of “hidden layers” to describe layers of nodes in neural networks that are neither inputs nor outputs.

To determine the outputs that a network computes from a specific set of inputs, one initializes the state of all gates to `false` and then iteratively and synchronously

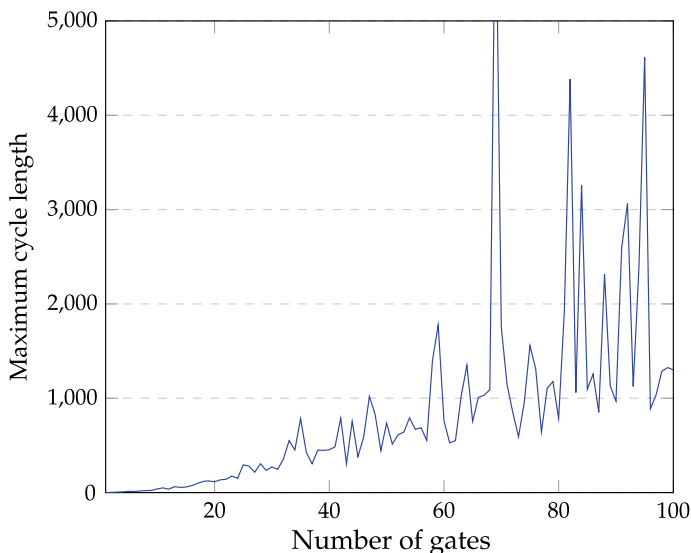


Fig. 11.4 Maximum cycle lengths for random networks of programmable gates. For each plotted point, 10,000 random networks of the specified size were created, initialized randomly, and run until a state was repeated

updates the state of all gates. Execution terminates when a state is repeated. If the values of output gates have remained unchanged throughout the attractor cycle into which the network has fallen, then those values are returned as the output from the computation. If they have changed over the course of the cycle, then the output of the computation is considered to be undefined and no value is returned.

11.4 Ratcheted Random Search

With the aim of advancing our understanding of the space of self-programming Boolean networks, I have chosen to search for networks that solve specified problems using extremely simple search methods.

Training cases are decomposed so that each case specifies the correct value for a single Boolean output. This means that if a problem involves u outputs then there will be u training cases for each collection of input values, each specifying the correct value of one of the outputs. For example, for the problem of finding a self-programming Boolean network that acts as a 2-bit $\times$ 2-bit multiplier, which must take four Boolean inputs and produce four Boolean outputs, there will be four training cases for each product. So for $2 \times 3 = 6$, which in binary is $10 \times 11 = 0110$, there will be a case specifying that the leftmost output should be 0 (*false*), another specifying that the next output should be 1 (*true*), and so on.

All of the methods discussed here maintain an initially empty collection of solved training cases and an initially complete collection of unsolved cases. At each step the current network (N) is tested on a randomly chosen unsolved case (C). If N produces the correct output for case C then C is moved from the unsolved collection to the solved collection. On the other hand, if it produces the incorrect output then a random change is made to N , producing N' . N' is then tested both on C and on all of the previously solved cases. If it produces the incorrect output for any of the previously solved cases then the change is reverted. That is, only steps in the search space that maintain correctness for all previously solved cases are allowed. This implements a “ratchet-like” mechanism that only permits changes that are neutral or better with respect to solved cases. If N' does still produce the correct outputs for all previously solved cases, and if it also produces the correct output for C , then the change is retained (N' is used as the next N) and C is moved from the unsolved to the solved collection. If N' produces the correct outputs for all previously solved cases but not for C then the question of whether to retain N' or revert to N is more complicated, as described below.

Although one might think at first that our quest for simplicity would best be served by a search method that is somehow “purely” random, it is not clear what that could or should mean in the context of the fact that there is no limit on network size. Most networks sampled from a purely random distribution will be bigger than our biggest computers, even though many problems that we care about can probably be solved with rather small networks. Because of this, and because we generally won’t know in advance how many gates are required to implement a target function, the search methods considered here all begin with a network that contains no hidden gates and allow changes to the number of gates to be among the random changes made during search.

Three kinds of network changes are permitted in the ratcheted random search algorithms presented here:

- Rewire: Change the source for a single input of a single gate, arbitrarily or locally (see below).
- Add: Add a new gate with all of its inputs set to random sources, and change one input of one pre-existing gate to use the output of the new gate as its source.
- Delete: Delete a randomly chosen gate and randomly choose another single source to replace all references to the deleted gate’s output.

If these kinds of changes are equally likely, and if all changes that are neutral with respect to solved cases are retained, then networks tend to grow rapidly and without bound at least for the problems studied here. Experiments have therefore been conducted with three settings that help to control network growth. In one (“add rarely”), the probability for choosing each kind of change is altered, typically by setting the probability of choosing Add to $\frac{1}{100}$ times the probability of choosing Rewire or Delete. In the second (“add for improvement”) and third (“add for disruption”), the conditions under which we retain the results of an Add are changed. Specifically, with “add for improvement”, the results of an Add are retained only if it correctly

solves not only all previously solved cases but also the current case C . In the “add for disruption” setting, although the results of the Add needn’t correctly solve C in order to be retained, the change must at least influence gate states during the processing of C . These variants are of course not the only possible mechanisms for controlling growth, but they are straightforward and not obviously biased. The first, changing the probability for additions relative to other network modifications, seems to control network growth reasonably well while still allowing solutions to be found in many settings. However, the introduction of probability parameters complicates analysis. The second approach, requiring the results of additions to solve C , has produced among the most concise solutions in some preliminary experiments but appeared to slow search and perhaps prevent the algorithm from ever finding solutions in some settings. The third approach, requiring that additions at least disrupt the processing of C , seems at the time of this writing to be the most generally promising of the variants considered here. Although further testing of these and other variants is warranted, we present here only results of the third approach.

Although the approaches described above help to focus search on reasonably small networks, even modestly sized networks have astronomically large numbers of rewiring options. Consider a problem with i inputs, so that a network with g gates would have $i + g + 2$ possible sources for each of $6g$ destinations. The number of possible wirings for this network is $(i + g + 2)^{6g}$, which is over 10^{73} for $i = 5$ and $g = 10$, and over 10^{180} for $i = 10$ and $g = 20$.

Based on considerations of the vastness of this space, and on observations of modular structure in complex systems, I have also experimented with two variants of Rewire changes. In one (“arbitrary rewiring”), the new source for the chosen input is picked from all possible sources, with each having equal probability. In the second (“local rewiring”), the source is chosen with uniform probability from a collection that contains one arbitrary source but also, if the current source is a gate rather than a network input or constant, all sources for all inputs to that gate. Because it has performed best in our preliminary investigations, I present here only results of runs with local rewiring.

I make no claim that the search methods used here will be the most effective for finding Boolean networks that solve specified problems, only that they are relatively simple and that, if they prove to be capable of finding solutions at all, then their simplicity may help us to understand the nature of the search space. For this reason I have tried to rely mostly on uniform random variation and the “ratchet” mechanism described above, making strategic modifications to this minimalist approach only to mitigate the effects of network growth, specifically by allowing additions only if they disrupt network dynamics and by preferring a kind of local rewiring. It may well be that more effective methods will utilize more sophisticated techniques, possibly based on traditional methods for digital circuit design or on work on evolvable hardware. For the present, however, I am aiming to deepen our understanding of the search space by relying on ratcheted random search as much as possible.

11.5 Results

To assess the scaling behavior of our search algorithms, experiments were conducted on search for even parity functions and multipliers, both of which have previously been studied as targets for the learning or evolution of combinational circuits.

For even parity, I searched 100 times for solutions to each of the even-2-parity, even-3-parity, even-4-parity, and even-5-parity problems. All searches found solutions, requiring the numbers of search steps shown in Fig. 11.5. Figure 11.6 shows the numbers of hidden gates in solution networks in those same runs.

Note that the number of search steps required to find a solution grows non-linearly with problem size. This is not surprising given the exponential growth in the state space as networks grow, but it does appear to show that more refined search methods will be required to solve larger problems, and it provides a benchmark that can be used in experiments with such methods. The growth in the number of hidden gates in solutions, however, appears to be more modest and approximately linear.

The search for multipliers is more challenging than the search for parity functions, in part because each computation produces multiple output bits. The algorithm described here can reliably find 3-bit $\times$ 3-bit multipliers, which have six input bits and six output bits, but these searches take long enough that statistics over large numbers of searches have not been gathered. That said, 3-bit $\times$ 3-bit multipliers

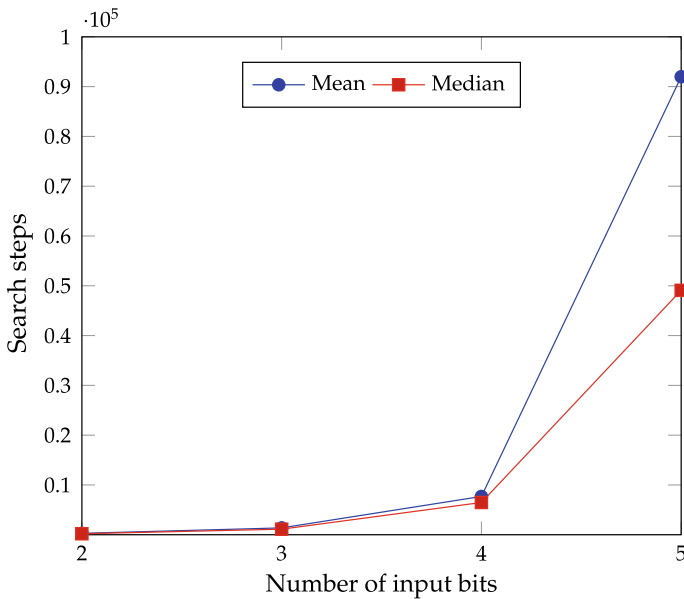
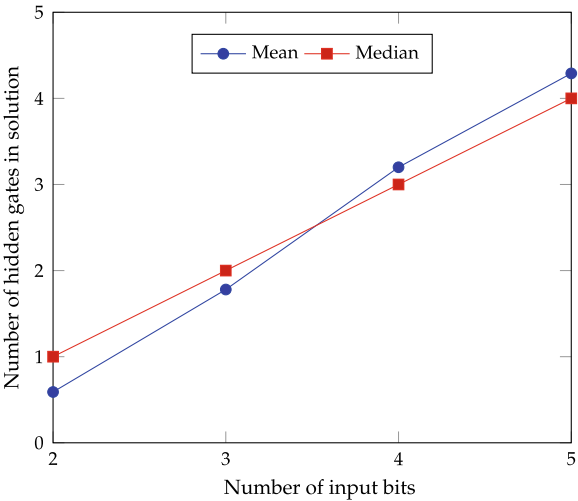


Fig. 11.5 Ratcheted random search steps required to find solutions to parity problems of the specified sizes, aggregated over 100 runs of each problem size

Fig. 11.6 Numbers of hidden gates in solutions to parity problems of the specified sizes, aggregated over the same 100 runs of each problem size used for Fig. 11.5



found by the algorithm can be interesting in structure and behavior, and one will be presented below.

For 2-bit $\times$ 2-bit multipliers, with four input bits and four output bits, I conducted 100 searches with the same algorithm as used above. All searches succeeded, requiring numbers of search steps with a mean of 16,952.51 and a median of 13,169.5. This means that for our algorithm finding a 2-bit $\times$ 2-bit multiplier requires roughly twice as many search steps as needed to find an even-4-parity function, but only roughly a quarter as many as needed to find an even-5-parity function. The mean number of hidden gates was 2.34 and the median was 2. These numbers are lower than those for even-4-parity, but the 2-bit $\times$ 2-bit multipliers include 4 output gates, in comparison to the 1 output gate for even-4-parity, so the total number of gates in 2-bit $\times$ 2-bit multipliers is higher.

I will use a 2-bit $\times$ 2-bit multiplier found by our algorithm to demonstrate some of the features of self-programming Boolean networks. A diagram of the multiplier that I will consider is shown in Fig. 11.7. Unfortunately, because of the large number of wires and the irregular connection pattern, it is difficult to glean much about how

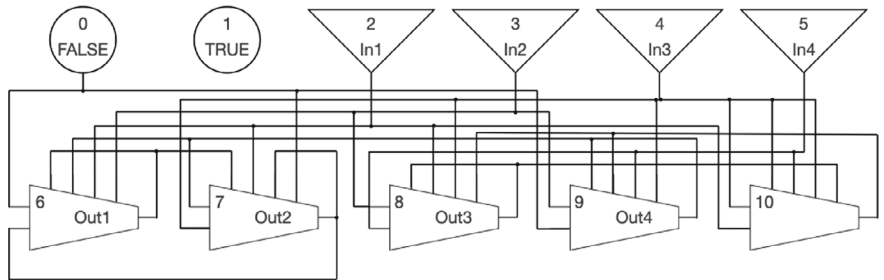


Fig. 11.7 A circuit diagram for a 2-bit $\times$ 2-bit multiplier found by the search algorithm presented here

a network functions from diagrams such as this one, even for such a small network. That said, it is clear from Fig. 11.7 that this multiplier makes no use of the constant `true` source and that all but one of its gates have self-loops, with the gate itself serving as one of its sources. One can also get a sense of the gate efficiency of the network by looking at it in comparison to diagrams of traditional combinational 2-bit $\times$ 2-bit multipliers that use 8 gates.

I have found it more useful to look at networks in tabular form in conjunction with execution traces for particular input patterns. Figure 11.8 shows such a trace for the multiplier in Fig. 11.7 when multiplying 3×3 .

The first line of the trace shows the indices for all of the network sources. The first two sources (with indices 0 and 1) are the constant `false` and `true` sources, respectively. The next four (2, 3, 4, and 5), following an empty column, are inputs; for this 2-bit $\times$ 2-bit multiplier the first number to be multiplied will be encoded in the two values appearing at indices 2 and 3, while the second number to be multiplied will be encoded in the values appearing at indices 4 and 5. The remaining indices all refer to programmable gates, with the output gates appearing first and then, after another empty column, any hidden gates. For all programmable gates the following six rows show the sources for the gates' inputs. The labels printed at the left of these rows show which input is which, with A and B indicating the sources for the inputs to the gate that will be programmed by the values at indices FF, FT, TF, and TT.

For example, consider the gate with index 6 in Fig. 11.8. The right-most column of the truth table for this gate will be whatever values are present at indices 6, 9, 2, and 3. If we suppose that there are `false` values at indices 6, 9, and 2, and a `true` value at index 3, then this will act as an AND gate on the values at indices 0 and 7. Incidentally, in this case that would mean that we already know that the gate's output will be `false`, because the 0 index always refers to the constant `false` and because a single `false` input is enough to guarantee a `false` output from an AND gate.

Starting on the eighth row of the trace in Fig. 11.8, following the rows for indices and sources of gate inputs, are rows showing the states of the network's sources (represented with `-` for `false` and `*` for `true` for the sake of legibility) and the hexadecimal representations of the logic functions that each gate is programmed by that state to perform. The first of these lines shows the network's initial state, in which the constant sources are `-` and `*` (as always), the four input values are all `*` (because both inputs are 3, which in binary is 11), and all other gates are initialized to `false`. The next line shows the programming of the gates as specified by the initial state. Specifically, the five gates are programmed to be hexadecimal 3, 3, 6, 3, and E, which by consulting Table 11.2 we can see are A , $B \wedge \neg A$, XOR, A , and NAND. The following row shows the state that results from the processing of these gates, in which we can see that the gates at indices 7 and 9 have flipped to `true`. Skipping to the bottom of the diagram we can see that the last two states are identical, which is why the computation terminated—in this case with a cycle length of 1, although that will not always be the case. We can also see that the values of the output gates in the final state are `true`, `false`, `false`, and `true`, which encode 1001 which is binary for 9, which is the correct answer for 3×3 . We can also see that along the way to

0	1	2	3	4	5	6	7	8	9	10	
						A	0	9	3	3	4
						B	7	4	5	0	2
						FF	6	6	8	9	4
						FT	9	2	2	10	5
						TF	2	7	4	5	4
						TT	3	0	10	4	8
—	*		*	*	*		—	—	—	—	—
							3	4	6	3	E
—	*		*	*	*		—	*	—	*	—
							7	6	6	B	E
—	*		*	*	*		*	—	—	*	—
							F	C	6	B	E
—	*		*	*	*		*	—	—	*	—

Fig. 11.8 A trace of the multiplier shown in Fig. 11.7 multiplying 3×3 . See text for description

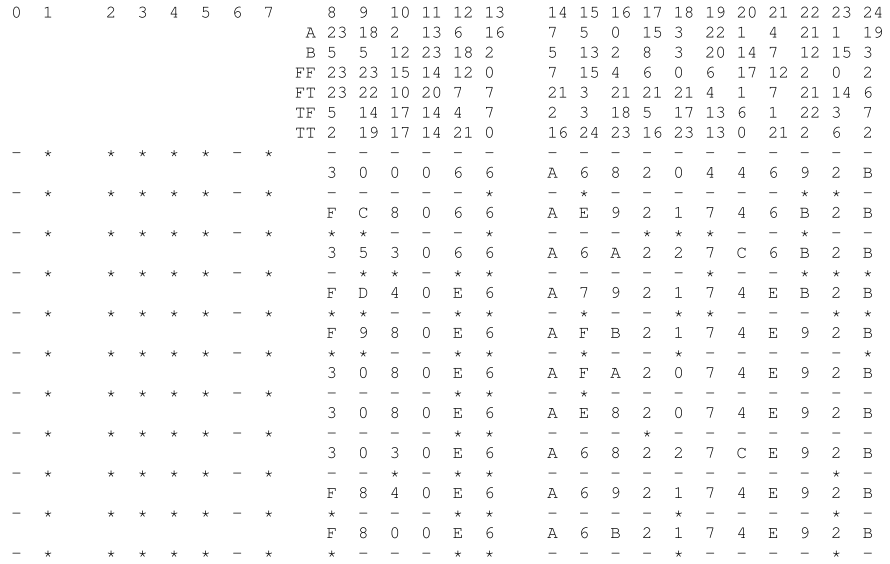
producing this result the state at some indices changed more than once, and several gates were repeatedly reprogrammed.

Figure 11.9 shows a trace of the same multiplier multiplying 1×0 . Here we can see that the initial gates immediately reproduce the initial state, causing immediate termination with the correct answer of 0. This illustrates the fact that a network may take different number of steps to produce a result for different inputs.

Figure 11.10 shows a trace of a network that our algorithm found for a 3-bit $\times$ 3-bit multiplier, shown multiplying 7×5 . Again we can see that the network is reasonably space efficient, using 17 programmable gates in comparison to the 40 gates used in a traditional combinational 3-bit $\times$ 3-bit multiplier. That said, networks that use as few as 15 programmable gates have also been found. We can also see again a variety of interesting structures, including self-loops, and the reprogramming of gates over the course of the computation.

0	1	2	3	4	5	6	7	8	9	10	
						A	0	9	3	3	4
						B	7	4	5	0	2
						FF	6	6	8	9	4
						FT	9	2	2	10	5
						TF	2	7	4	5	4
						TT	3	0	10	4	8
—	*	—	*	—	—	—	—	—	—	—	—
							1	0	0	0	0
—	*	—	*	—	—	—	—	—	—	—	—

Fig. 11.9 A trace of the multiplier shown in Fig. 11.7 multiplying 1×0 . See text for description



of random moves that are considered during search, asking for example whether search will be more efficient if different probabilities of change are applied to data inputs (A and B) than to programming inputs (FF, FT, TF, and TT). A variety of other analyses of the experimental data may also be revealing, ranging from analysis of variance of the measures that have already been presented to the calculation of measures of chaotic dynamics. Additional control experiments might also be conducted, for example with the ratchet mechanism disabled or with programming inputs wired only to constants. The effects of other constraints on wiring patterns, such as restriction to a 3D lattice, might also be considered. Comparisons and/or synergies with other work on search for executable networks might be explored, for example with recurrent Cartesian genetic programming [17] or Markov Brains [6]. Finally, it would be interesting to examine this work in the context of studies of other computational substrates that involve complex underlying dynamics, for example with physical reservoir computing [14].

Acknowledgements Many thanks to Bill Tozier, Mahran Yousef, members of the PUSH lab at Amherst College, and the University of Massachusetts, Amherst. Thanks also to participants in the 2024 Genetic Programming Theory and Practice workshop, especially including Wolfgang Banzhaf, Alex Lalejini, Penousal Machado, Charles Ofria, Bill Tozier, and Leonardo Vanneschi. This work was performed in part using high-performance computing equipment obtained under National Science Foundation Grant No. 2117377. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Competing Interests The author has no conflicts of interest to declare that are relevant to the content of this chapter.

References

1. Ali, B., Almaini, A.E.A., Kalganova, T.: Evolutionary algorithms and theirs use in the design of sequential logic circuits. *Genet. Program Evolvable Mach.* **5**, 11–29 (2004)
2. East, R., Rowe, J.: Abstract genetic representation of dynamical neural networks using Kauffman networks. *Artif. Life* **3**(2), 67–80 (1997)
3. Gershenson, C.: Introduction to Random Boolean Networks (2004). <https://api.semanticscholar.org/CorpusID:5278615>
4. Gershenson, C.: Guiding the self-organization of random Boolean networks. *Theory Biosci.* **131**(3), 181–191 (2012)
5. Haddow, P.C., Tyrrell, A.M.: Evolvable hardware challenges: past, present and the path to a promising future. In: Stepney, S., Adamatzky, A., (eds.), *Inspired by Nature. Emergence, Complexity and Computation*, vol. 28. Springer, Cham (2018)
6. Hintze, A., Edlund, J.A., Olson, R.S., Knoester, D.B., Schossau, J., Albantakis, L., Tehrani-Saleh, A., Kvam, P., Sheneman, L., Goldsby, H., Bohm, C., Adami, C.: Markov brains: a technical introduction (2017). <https://doi.org/10.48550/arXiv.1709.05601>
7. Kauffman, S.: Homeostasis and differentiation in random genetic control networks. *Nature* **224**, 177–178 (1969)
8. Kauffman, S.: *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, New York (1993)
9. Kauffman, S.: *At Home in the Universe*. Oxford University Press, New York (1995)

10. Lemke, N., Mombach, J.C.M., Bodmann, B.E.J.: A numerical investigation of adaptation in populations of random boolean networks. *Phys. A* **301**, 589–600 (2021)
11. Mazare, A., Ionescu, L., Serban, G., Barbu, V.: Evolvable hardware with Boolean functions network implementation. In: 2011 International Conference on Applied Electronics Applied Electronics (AE). IEEE (2011)
12. Miller, J. (ed.): *Cartesian Genetic Programming*. Springer, Berlin (2011)
13. Miller, J.F.: Cartesian genetic programming: its status and future. *Genet. Program Evolvable Mach.* **21**, 129–168 (2020)
14. Nakajima, K.: Physical reservoir computing—an introductory perspective. *Japanese J. Appl. Phys.* **59**(6) (2020)
15. Tetteh, M., Dias, D.M., Ryan, C.: Grammatical evolution of complex digital circuits in system Verilog. *SN Comput. Sci.* **3**, 188 (2022)
16. Shanthi, A.P., Parthasarathi, R.: Practical and scalable evolution of digital circuits. *Appl. Soft Comput.* **9**, 618–624 (2009)
17. Turner, A.J., Miller, J.F.: Recurrent cartesian genetic programming. In: Bartz-Beielstein, T., Branke, J., Filipič, B., Smith, J. (eds.), *Parallel Problem Solving from Nature – PPSN XIII*. PPSN 2014. Lecture Notes in Computer Science, vol. 8672. Springer, Cham (2014)
18. Vasicek, Z.: Bridging the gap between evolvable hardware and industry using cartesian genetic programming. In: Stepney, S., Adamatzky, A., (eds.), *Inspired by Nature. Emergence, Complexity and Computation*, vol. 28. Springer, Cham (2018)
19. Walker, J.A., Miller, J.F.: Improving the evolvability of digital multipliers using embedded cartesian genetic programming and product reduction. In: Moreno, J.M., Madrenas, J., Cosp, J., (eds.), *Evolvable Systems: From Biology to Hardware*. ICES 2005. Lecture Notes in Computer Science, vol. 3637. Springer, Berlin (2005)
20. Wikipedia (2024) Multiplexer. https://en.wikipedia.org/wiki/Multiplexer#Digital_multiplexers
21. Wuensche, A.: The ghost in the machine: basins of attraction of random boolean networks. In: *Artificial Life III Proceedings*, Santa Fe Institute Studies in the Sciences of Complexity (1994)

Chapter 12

Exploring Non-bloating Geometric Semantic Genetic Programming



Leonardo Vanneschi, Davide Farinati, Diogo Rasteiro, Liah Rosenfeld, Gloria Pietropolli, and Sara Silva

Abstract Recently, a new variant of Geometric Semantic Genetic Programming (GSGP) was introduced that, while maintaining the property of inducing a unimodal error surface for all supervised learning problems, is able to generate models that are compact enough to be interpretable by humans. This variant is called the Semantic Learning algorithm based on Inflate and deflate Mutation (SLIM_GSGP) and, as the name suggests, it is based on two types of mutation: one (inflate) that generates offspring larger than their parents, similar to traditional geometric semantic mutation, and the other (deflate) that generates offspring smaller than their parents. This chapter reviews and extends the initial work on SLIM_GSGP by introducing two novel variants, thoroughly studying the geometric characteristics of the SLIM_GSGP operators and discussing the experimental results and their interpretation in greater depth. The main conclusion is that SLIM_GSGP is a very promising method, warranting significant investment in future research.

L. Vanneschi (✉) · D. Farinati · D. Rasteiro · L. Rosenfeld
NOVA Information Management School (NOVA IMS), Universidade Nova de Lisboa, Campus de Campolide, 1070-312 Lisboa, Portugal
e-mail: lvanneschi@novaims.unl.pt

D. Farinati
e-mail: dfarinati@novaims.unl.pt

D. Rasteiro
e-mail: drasteiro@novaims.unl.pt

L. Rosenfeld
e-mail: lrosenfeld@novaims.unl.pt

G. Pietropolli
Department of Mathematics, Informatics, and Geosciences, University of Trieste, Trieste, Italy
e-mail: gloria.pietropolli@phd.units.it

S. Silva
LASIGE, Department of Informatics, Faculty of Sciences, University of Lisbon, Lisbon, Portugal
e-mail: sgsilva@fc.ul.pt

12.1 Introduction

Geometric Semantic Genetic Programming (GSGP) [18, 27] is a variant of Genetic Programming (GP) [14, 23] that uses specific genetic operators, termed Geometric Semantic Operators (GSOs), in lieu of the traditional GP crossover and mutation. Since its inception in 2012 [18], various developments have significantly enhanced GSGP [3–6, 8, 9, 12, 13, 15, 17, 19, 20, 22, 30], leading to notable achievements across diverse application domains [7, 10, 16, 29, 33]. However, along with its primary advantage, GSGP has also maintained a significant drawback. The advantage lies in its ability to induce a unimodal error surface for each supervised learning problem, endowing the algorithm with highly competitive training data optimization capabilities. The drawback stems from the fact that GSOs consistently generate offspring larger than their parents, resulting in a continuous and rapid growth in the size of the individuals. Consequently, at the end of a GSGP run, the returned model is typically too large to be interpreted by humans.

In 2024, a new system of GSGP was introduced [28], aiming to preserve the ability to induce a unimodal error surface, while evolving models that are compact enough for human interpretation. This system, named Semantic Learning algorithm based on Inflate and deflate Mutation (SLIM_GSGP), incorporates alongside the traditional Semantic Geometric Mutation (GSM), termed inflate mutation due to its tendency to generate offspring larger than their parents, a new geometric semantic mutation named deflate mutation. The deflate mutation generates offspring smaller than their parents by removing part of the parent genotype while preserving all the geometric semantic properties of traditional GSM. Particularly, like traditional GSM, deflate mutation induces a ball mutation in the semantic space, of a radius equal to the value of a parameter, the mutation step ms , consequently leading to a unimodal error surface. In [28], four variants of SLIM_GSGP (simply called SLIM from now on) were defined by combining two different methods of obtaining a function with values in $[-ms, ms]$ with two different methods of obtaining a ball mutation in the semantic space. This chapter introduces one additional method of obtaining a function with values in $[-ms, ms]$ and studies the six distinct resulting variants of SLIM, four of which are those introduced in [28]. The exploration of the six variants prompts an examination of the dynamics of operators across all variants, aiming to understand the similarities and differences between them. Consequently, this manuscript also delves into the geometry of the ball defined by the various types of semantic mutations used. This analysis not only clarifies the functioning of the different SLIM variants but also deepens the understanding of why it is often advantageous in GSGP to employ a random mutation step, generated uniformly from a predefined interval at each mutation event. Previous works [29, 33] have shown the utility of this practice, yet failed to provide a thorough explanation of the underlying rationale.

In summary, the contributions of this work are the introduction of two novel variants of SLIM, a full and deeper exploration of the results of all six variants and

an in-depth examination of the geometry of the operators used across all the presented SLIM variants.

The document is organized as follows: Sect. 12.2 briefly introduces traditional GSGP. Section 12.3 discusses the main intuition that has led to the definition of SLIM. In Sect. 12.4 we discuss the different SLIM variants. In Sect. 12.5 we investigate the geometry of the different SLIM operators. In Sect. 12.6 we present our experimental study. Finally, Sect. 12.7 concludes the paper and suggests ideas for future research.

12.2 Geometric Semantic Genetic Programming

Let $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$ be a collection of input data, which can be interpreted as the training examples (observations or fitness cases) of a supervised learning task. Let $\mathbf{t} = [t_1, t_2, \dots, t_n]$ be the vector of the expected outputs (or target values) associated with these inputs. A GP individual (or program) P can be seen as a function that takes $\mathbf{x}_i$ as its input and produces an output, which we denote as $P(\mathbf{x}_i)$. As in [18], we refer to the vector $s_P = [P(\mathbf{x}_1), P(\mathbf{x}_2), \dots, P(\mathbf{x}_n)]$ as the *semantics* of P . These semantics can be visualized as a point within an n -dimensional space, referred to as the *semantic space*.

As mentioned earlier, GSGP is a variant of GP where Geometric Semantic Operators (GSOs) take the place of the conventional crossover and mutation. GSOs introduce alterations to the syntax of GP individuals, aiming for a controlled and quantifiable effect on their semantics. In the last decade, numerous contributions have shown that mutation is a more powerful operator than crossover for GSGP, with mutation-only GSGP returning results that are comparable, or even better than the ones obtained when a mixture of crossover and mutation is used [5, 33]. For this reason, in this work we only consider Geometric Semantic Mutation (GSM). According to the definition of GSM¹ given in [18], we have the following.

Geometric Semantic Mutation (GSM). Given a parent function $T : \mathbb{R}^n \rightarrow \mathbb{R}$, GSM with mutation step ms returns the function, $GSM(T) = T + ms \cdot (T_{R1} - T_{R2})$, where T_{R1} and T_{R2} are random functions.

The motivation for considering the difference between two random functions, T_{R1} and T_{R2} , is that GSM should introduce a random perturbation of T with distribution centered at zero. The difference between two random expressions is a random expression that *a priori* has equal chances of being positive or negative, so its distribution is centered at zero. Also, while this was not originally defined in GSM, later studies [13, 27, 33] have demonstrated that limiting the codomain of T_{R1} and T_{R2} to a predefined interval such as $[0, 1]$ (e.g., by applying a sigmoid or akin function to their outputs [13]) greatly improves the generalization ability of GSGP. In this way, the application of GSM to a given parent program produces a random perturbation

¹ Here we define GSM only for symbolic regression problems. For definitions of GSOs in other domains, readers are referred to [18].

of its semantics; such a perturbation follows a zero-centered distribution in a user-controlled range $[-ms, ms]$, where ms (the mutation step) is a hyperparameter. In other words, GSM corresponds to a *ball mutation* of radius ms in the semantic space [18] and it induces a unimodal error surface for any supervised learning problem: the error surface (on training data) has no locally optimal solution except for the semantically unique global optimum. This holds for any supervised learning problem, independently from, for instance, the size and complexity of the data. For a detailed explanation of GSGP and thorough justifications of the fact that the fitness landscape induced by GSM is indeed unimodal, the reader is referred to [18, 27, 32].

12.3 The Basic SLIM_GSGP Intuition

Given that $GSM(T) = T + ms \cdot (T_{R1} - T_{R2}) = T - ms \cdot (T_{R2} - T_{R1})$ and given that T_{R1} and T_{R2} can be thought as independent random variables with the same probability distribution over the space of expressions, T_{R1} and T_{R2} are interchangeable. In other words, if the definition of GSM was $GSM(T) = T - ms \cdot (T_{R1} - T_{R2})$, then this definition would be equivalent to the one given in the previous section. Considering this, nothing prevents us from applying GSM by alternating the use of these two different forms. Now, let us consider a chain of, for instance, three mutation events starting from an individual T and let us assume that, in these three mutation events, we use the original definition of GSM

$$GSM(T) = T + ms \cdot (T_{R1} - T_{R2}) \quad (12.1)$$

$$GSM(GSM(T)) = T + ms \cdot (T_{R1} - T_{R2}) + ms \cdot (T_{R3} - T_{R4}) \quad (12.2)$$

$$GSM(GSM(GSM(T))) = T + ms \cdot (T_{R1} - T_{R2}) + ms \cdot (T_{R3} - T_{R4}) + ms \cdot (T_{R5} - T_{R6}) \quad (12.3)$$

If, at the next step, we use instead subtraction, we obtain an individual like

$$T + ms \cdot (T_{R1} - T_{R2}) + ms \cdot (T_{R3} - T_{R4}) + ms \cdot (T_{R5} - T_{R6}) - ms \cdot (T_{R7} - T_{R8})$$

Clearly, the genotype of this individual is larger than the one in Eq. (12.3). In other words, using subtraction instead of addition allowed us to apply a mutation that has exactly the same geometric properties as the GSM defined in Sect. 12.2, but was not helpful in reducing the size of the generated individual. But what if instead of subtracting a new term, we had subtracted a term that had been previously added? For instance, what if the term that we subtract from the individual in Eq. (12.3) is, say, $ms \cdot (T_{R3} - T_{R4})$? In the end, the idea of reusing random trees instead of generating new ones every time they are needed is not new (it has already been employed, for instance, in [30]). In that case, that term could be simplified, generating an individual with a smaller size:

$$\begin{aligned}
T + ms \cdot (T_{R1} - T_{R2}) &+ \cancel{ms \cdot (T_{R3} - T_{R4})} + ms \cdot (T_{R5} - T_{R6}) - \cancel{ms \cdot (T_{R3} - T_{R4})} \\
&= T + ms \cdot (T_{R1} - T_{R2}) + ms \cdot (T_{R5} - T_{R6})
\end{aligned} \tag{12.4}$$

After simplification, obviously, the individual in Eq. (12.4) has a smaller genotype than the one in Eq. (12.3). However, it is important to reinforce two statements. First, subtracting a term that had previously been added is *not* a backtracking operation. In other words, it generally does not generate an individual that had already been in the population in the previous iterations. This can be seen by observing the sequence of mutation events reported in Eqs. (12.1), (12.2) and (12.3). Notice that the individual in Eq. (12.4) was not in that sequence (this is general, unless the subtracted term is the last one that had been added). Second, subtracting a term that had previously been added has the same type of effect as GSM: a perturbation of each semantic coordinate by a quantity in $[-ms, ms]$, and therefore it is also a geometric semantic mutation. This is the idea at the basis of SLIM. As the name suggests, SLIM uses a mixture of two types of mutation. The first one generates an offspring that is larger than its parent, as traditional GSM does, which is called inflate mutation. The other produces an offspring that is smaller than its parent, exploiting the idea explained above, and it is called deflate mutation. SLIM, however, can have different variants, that are presented in the continuation.

12.4 The Different SLIM_GSGP Variants

We study three different ways of defining a function that returns values in $[-ms, ms]$ and two different ways of perturbing an individual to obtain ball mutation on the semantic space. Combining these options, we obtain six variants of SLIM.

Different ways of defining a function with values in $[-ms, ms]$. As seen in traditional GSM, such a function can be the difference between two random trees T_{R1} and T_{R2} , each wrapped in a sigmoid-like function S that constrains its outputs to $[0, 1]$, multiplied by the mutation step ms . We call this function SIG2, since it uses a sigmoid (SIG) to wrap each of the two (2) random trees:

$$SIG2 = ms \cdot (S(T_{R1}) - S(T_{R2})) \tag{12.5}$$

However, as already pointed out in the literature [3, 16], it is possible to use only one random tree T_R , instead of two, to define a function with outputs in $[-ms, ms]$, and this allows us to save a significant amount of computational resources. The approach presented in [16] solves the problem using mutation steps extracted from a previously decided distribution. The first single-tree approach used here is inspired by [3] (where also different types of normalizations were implemented), and we call it ABS because it uses the absolute value to normalize the outputs of the random tree:

$$ABS = ms \cdot (1 - \frac{2}{1 + |T_R|}) \quad (12.6)$$

Even though, as discussed in [3], for some values of T_R this expression may not be zero-centered (in other words, the distribution of its output values can be biased toward specific directions), still the effectiveness of using this type of normalization with GSM was demonstrated in [3].

The second approach to defining a function in $[-ms, ms]$ using only one random tree is novel. We call it SIG1 because it also uses the sigmoid (SIG) to constrain the outputs of the single (1) random tree:

$$SIG1 = ms \cdot (2 \cdot S(T_R) - 1) \quad (12.7)$$

Different ways of obtaining ball mutation on the semantic space. A small perturbation of a numeric value can trivially be obtained either by adding a positive or negative quantity that is close to zero or by multiplying a quantity that is close to (i.e., either slightly smaller or slightly larger than) one. Therefore, the inflate mutation can either add the SIG2, ABS or SIG1 functions defined in Eqs. (12.5), (12.6) and (12.7), or it can multiply by $1 + \text{SIG2}$, $1 + \text{ABS}$ or $1 + \text{SIG1}$, respectively. In the variants where inflate adds, deflate subtracts a term that had previously been added; in the variants where inflate multiplies, deflate divides a term that had previously been multiplied.

Combining the three ways of defining a function in $[-ms, ms]$ and the two ways of obtaining ball mutation, we obtain six SLIM variants, from now on denoted as SLIM+ or SLIM* (depending on whether they use addition or multiplication) followed by SIG2, ABS or SIG1 (depending on which function they use).

Linked-List Implementation of SLIM_GSGP. All the SLIM variants have been implemented by representing the genotypes as linked lists. In this way, an inflate mutation can be implemented by a simple append of an item at the end of the list, while a deflate mutation can be implemented by a simple delete of one of the items of the list. Finally, an individual can be evaluated simply by accumulating the sum (for SLIM+SIG2, SLIM+ABS and SLIM+SIG1) or the product (for SLIM*SIG2, SLIM*ABS and SLIM*SIG1) of the terms in the list. An example of a SLIM+SIG2 genotype and the individuals resulting from the application of an inflate followed by a deflate is shown in Fig. 12.1.

The head of the list contains the initial tree T , created at population initialization. Inflate mutation can always be applied, and it simply appends a new list item. Deflate mutation can be applied only if the length of the list is strictly larger than one (i.e., deflate can be applied only to individuals that, besides the head of the list, contain at least another item), and it deletes one random item that is different from the head.

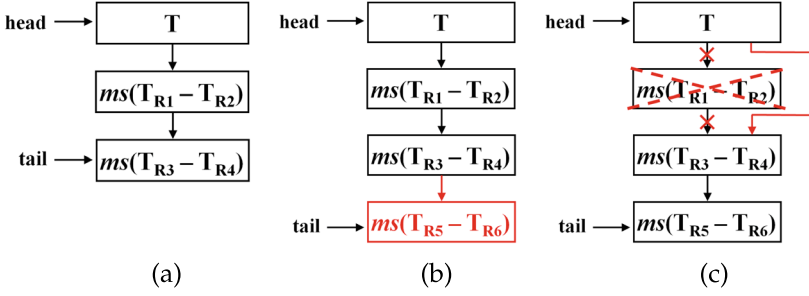


Fig. 12.1 An example genotype of a SLIM+SIG2 individual (a), after an inflate mutation (b) and followed by a deflate mutation (c)

12.5 Diving into the Geometry of the SLIM_GSGP Variants

In previous sections, we discussed how geometric semantic mutations represent the concept of a ball mutation in the semantic space. However, the geometry of these induced balls varies across the different operators. These differences can potentially impact the dynamics of each operator and, consequently, the overall performance of the algorithm. We now visualize and discuss the geometry of the various SLIM operators. To achieve this, we consider the simplified scenario of semantic vectors of length 2, representing a regression problem instance with a dataset containing only two observations. Therefore, each individual is a point in a 2-dimensional space. We arbitrarily set the semantics of an individual T to (10, 10) and then plot the semantics of a sample of 5000 neighbors of T (i.e., 5000 individuals resulting from the mutation of T) using the different mutation operators. This produces a scatterplot showing the distribution of all these offspring relative to the parent. In these experiments, the output of the random trees used by the operators is simulated by random numbers drawn uniformly from an interval. We use three different intervals, $[-1, 1]$, $[-10, 10]$ and $[-100, 100]$, to study the impact caused by the magnitude of the outputs of the random trees. We also use two different settings for the mutation step, a fixed $ms = 1.0$ and a random ms , drawn uniformly from $[0, 1]$ at every mutation event.

SLIM+SIG2. Figure 12.2 reports the scatterplots for SLIM+SIG2. The first (respectively, the second) line of scatterplots reports the dynamics of the inflate (respectively, deflate) mutation for a constant ms . Analogously, the third (respectively, the fourth) line reports the dynamics of the inflate (respectively, deflate) mutation when a random ms is employed. The first, second and third columns report the results when the output of the random trees is within $[-1, 1]$, $[-10, 10]$ and $[-100, 100]$, respectively. In all the scatterplots, the point (10, 10) (representing the semantics of the parent) is represented in red and the other points (semantics of the offspring) in black.

An observation of the first two lines of scatterplots immediately gives a visual intuition of the influence of the range of variations of the outputs of the random

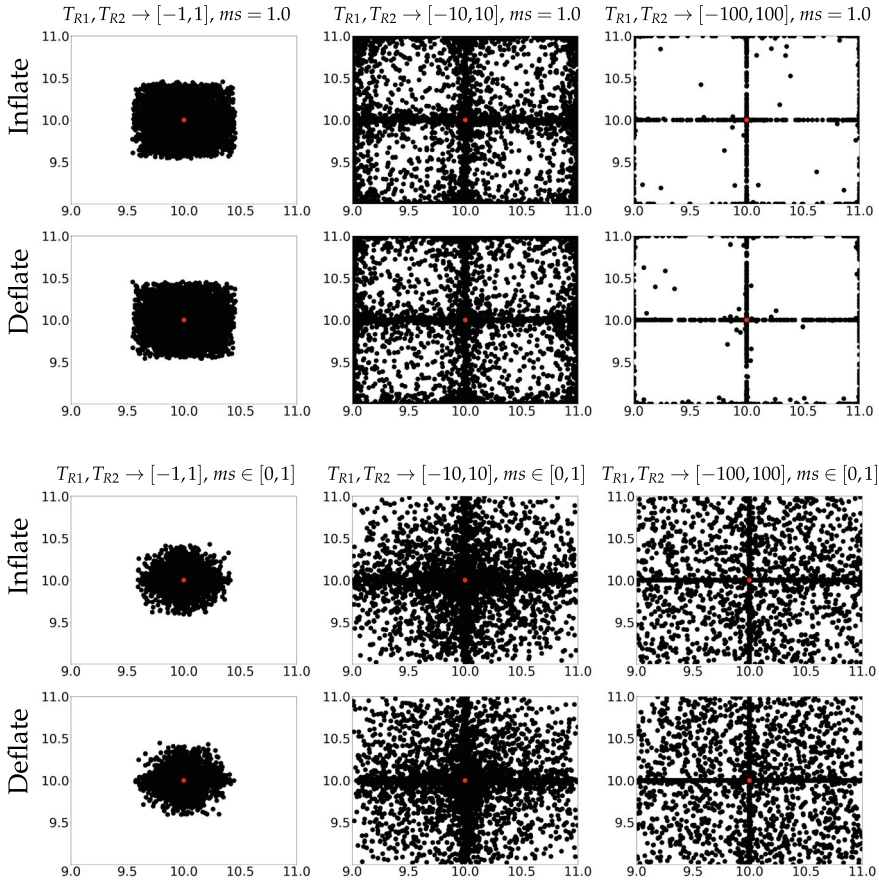


Fig. 12.2 Geometry of SLIM+SIG2 in a 2-dimensional semantic space. The red point represents the parent semantics and the black points represent a sample of 5000 offspring. The first and second rows of plots represent inflate and deflate mutation with $ms = 1.0$, while the third and fourth rows represent inflate and deflate mutation with random $ms \in [0, 1]$. The three columns represent the results when the outputs of the random trees are in $[-1, 1]$, $[-10, 10]$ and $[-100, 100]$

trees in shaping the geometry of the ball induced by the SLIM+SIG2 operators. As expected, inflate and deflate mutation of SLIM+SIG2 induces a practically identical ball. In the case of constant ms , the ball has a radius approximately equal to ms . However, when the outputs of the random trees are values in $[-1, 1]$, the ball is dense, containing points almost everywhere possible, while for $[-10, 10]$ the ball becomes less dense, with areas that are not reached, until the last interval $[-100, 100]$ where we can see that, except for some outliers, only the values 9.0, 10.0 and 11.0 are reached, corresponding to a perturbation of the coordinates equal to $-ms$, 0 and ms . This is a clear effect of using the sigmoid to normalize the output of the random trees: while the sigmoid returns values that span rather uniformly the interval $[-ms, ms]$

when its inputs are in $[-1, 1]$, it tends to “flatten” its outputs toward its asymptotes when the inputs have a larger magnitude. Under this perspective, the third and fourth lines of the figure allow us to appreciate the importance of using a random ms . When the outputs of the random trees assume values in $[-1, 1]$, the ball looks smaller than the corresponding case of constant ms , and more concentrated toward its center. However, when the outputs of the random trees assume values from larger intervals, having a random ms still produces a rather dense ball with coordinate values that, despite being more concentrated on 9.0, 10.0 and 11.0, also spread inside the ball, even when using the largest interval $[-100, 100]$. It is important that mutation is able to generate those intermediary values, particularly when one or more coordinates are close to the corresponding target. These scatterplots corroborate a fact already observed in the literature: GSGP with random ms usually outperforms GSGP with constant ms [13, 33].

SLIM*SIG2. The scatterplots of SLIM*SIG2 are reported in Fig. 12.3. Comparing this figure with the previous ones, we can observe that the inflate mutation of SLIM*SIG2 induces a ball of approximately the same shape (although larger²) as the inflate mutation of SLIM+SIG2, for both fixed and random ms . However, the deflate mutation of SLIM*SIG2 induces a ball that is not equally distributed around the parent. We observe no coordinate values lower than 5, a limit that stems directly from the definition of SIG2 (Eq. (12.5)) and the fact that the SLIM*SIG2 deflate operator is a division.³ Furthermore, for the larger intervals $[-10, 10]$ and $[-100, 100]$, many points have extreme (positive) values in one of the coordinates. They are not shown because they are way beyond the scale used in the plot axes. This happens because small values in the denominator naturally lead to extreme outputs. As a consequence, SLIM*SIG2 can perform not only small perturbations of the coordinates, but also large perturbations that allow the algorithm to quickly “jump” to a different area of the semantic space.

SLIM+ABS and SLIM*ABS. The scatterplots of SLIM+ABS are reported in Fig. 12.4, where we observe different dynamics from the previous variants. The location of the points is constrained by the limits imposed by the ABS formula, with a distribution that is never centered around the parent. Another distinctive characteristic of this variant is the complementarity between the inflate and deflate operators, whose distributions appear symmetrical. Although promising results have already been obtained using only the inflate mutation of SLIM+ABS [3], this complementarity suggests that both inflate and deflate operators should be used. On the other hand, we also notice similarities with the previous variants. Increasing the output interval of the random trees causes the points to concentrate more on certain edges, an effect that was also present in the previous variants; the larger the interval, the more concentrated the points become. In SLIM+ABS, this problem is particularly visible with the $[-100, 100]$ interval when $ms = 1.0$, as the inflate mutation can no

² Remark the different scales on the axes of Fig. 12.3 compared to Fig. 12.2.

³ The maximum value of the sigmoid is 1, so the maximum value of the denominator is 2. Therefore, considering the parent (10, 10), the minimum result of the division is 5.

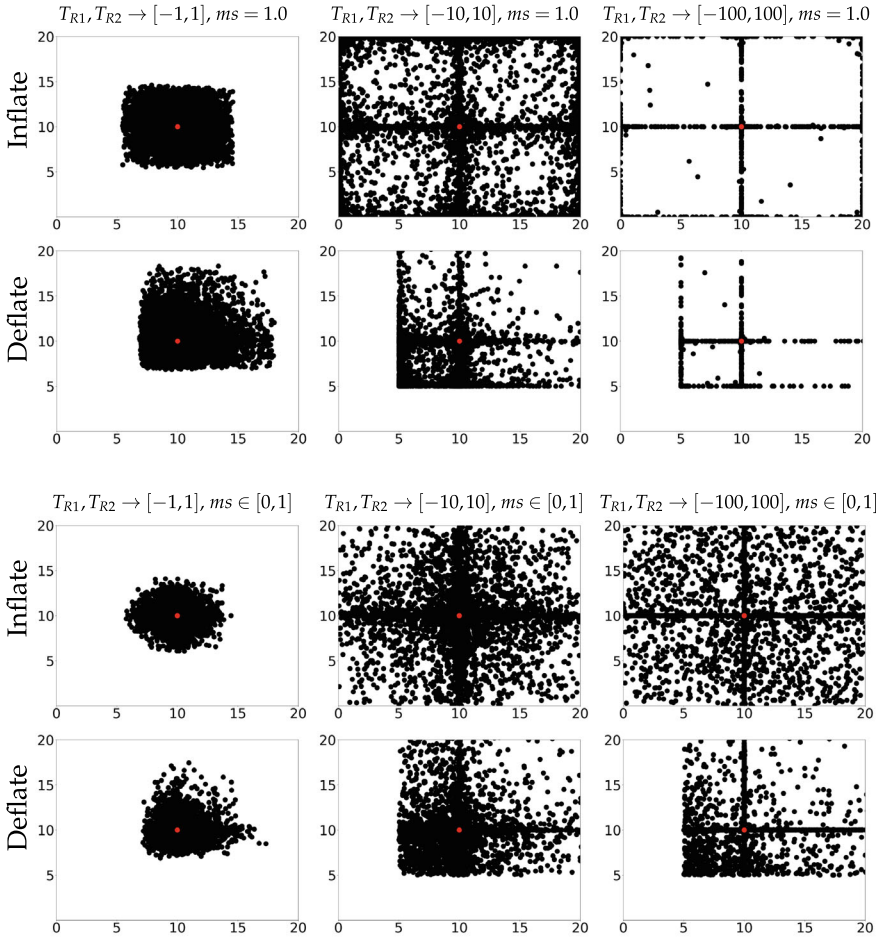


Fig. 12.3 Geometry of SLIM*SIG2. Plots like in the previous figure

longer generate small perturbations of the parent's semantics. The random ms again, like in the previous variants, avoids this extreme concentration and spreads the values more evenly across a larger area, except for the interval $[-1, 1]$ that actually shrinks the area.

For SLIM*ABS, we do not show the plots to save space. They are similar to the ones of SLIM+ABS in the shapes and complementarity between inflate and deflate, as well as in the observed effects of the different intervals and mutation steps. They are different from SLIM+ABS, but similar to SLIM*SIG2 in that deflate never produces values below 5 but produces many points with extreme values in one of the coordinates. SLIM*ABS induces larger mutation balls than SLIM+ABS, allowing for larger jumps in the semantic space (the same way SLIM*SIG2 induces larger mutation balls than SLIM+SIG2).

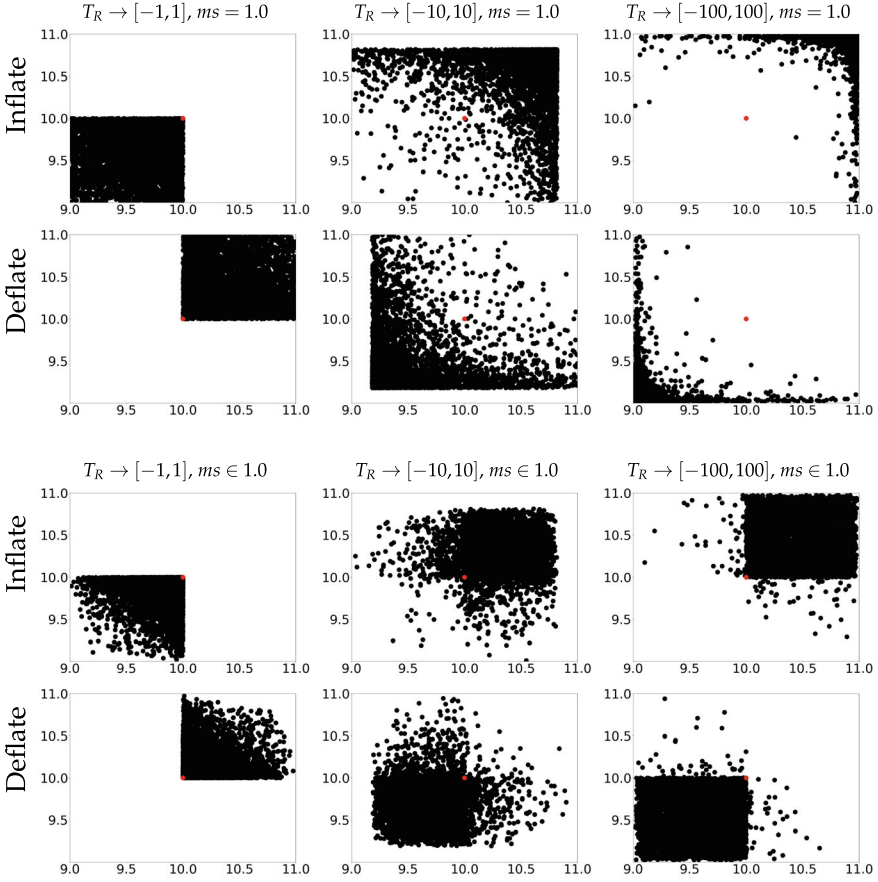


Fig. 12.4 Geometry of *SLIM+ABS*. Plots like in the previous figure

SLIM+SIG1 and SLIM*SIG1. The scatterplots of *SLIM+SIG1* are reported in Fig. 12.5. For the interval $[-1, 1]$ of the random trees, the behavior is similar to the one observed in *SLIM+SIG2*, with an even spread of points around the parent. The random ms achieves a better distribution than *SLIM+SIG2*, particularly for deflate. For the larger intervals with $ms = 1.0$, we observe a strong concentration of points on the edges. Unlike *SLIM+SIG2*, where points concentrated also around each one of the parent's coordinates, and unlike *SLIM+ABS*, where the distributions of inflate and deflate were both skewed in one direction and complemented each other, here the points are evenly pushed away from the parent, equally for inflate and deflate. For a random ms , this problem is solved and the points nearly cover the entire area of the ball. Again, this is an indication of how advantageous it is to use a random mutation step.

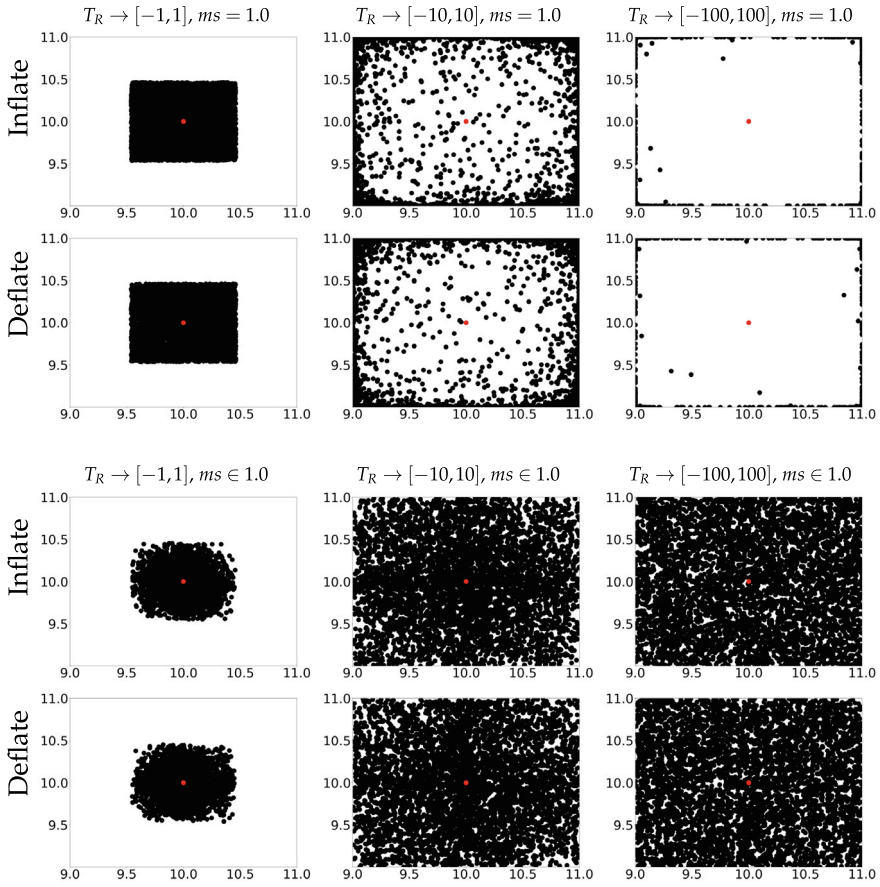


Fig. 12.5 Geometry of *SLIM+SIG1*. Plots like in the previous figure

For *SLIM*SIG1*, we do not show the plots. They are similar to the ones of *SLIM+SIG1* while exhibiting the same effects observed when comparing *SLIM+SIG2* with *SLIM*SIG2*. They generate larger mutation balls than *SLIM+SIG1* and the deflate produces skewed distributions with extreme positive values in one of the coordinates, and no values below 5.

12.6 Experimental Study

12.6.1 Test Problems and Experimental Settings

As in [28], the test problems used in this work are six real-life symbolic regression applications that have often been used as benchmarks in the GP literature (e.g., [2, 10, 21, 25, 31]): Toxicity (626 features and 274 samples; see [2]); Istanbul (7 features and 536 samples; see [1]); Energy (8 features and 330 samples, selected from the dataset presented in [26]); PPB (626 features and 234 samples; see [2]); Concrete (8 features and 1030 samples; see [10]) and Residential Buildings Sales Price (RBSP) (107 features and 372 samples; see [24]).

On these problems, we compared standard GP (STDGP) and GSGP with the SLIM variants, in a total of eight methods. We did Monte Carlo cross-validation [11, 32] by making 30 different random data partitions into training (70%) and test (30%) sets. All methods did 30 runs, each using one of the 30 partitions.

The following settings are the same used in [28]. The trees were initialized with maximum depth 6 using ramped half-and-half. The function set contained the four binary arithmetic operators $+$, $-$, $*$ and $/$ protected as in [14], whereas the terminal set contained only the variables of the problem, without any constants. The population size was 100 individuals, allowed to evolve for 2000 generations. Fitness was the root mean squared error (RMSE) between predicted and expected outputs. Selection was performed with tournaments of size 2 and elitism was used, copying the best individual of each population to the next generation.

Specifically for STDGP, it used standard subtree crossover and mutation with probabilities 0.8 and 0.2, respectively, and a maximum tree depth of 17. GSGP used only its GSM operator, no crossover (as suggested in [19, 33]) and a random mutation step drawn uniformly from $[0, 1]$ in each mutation event (as in [33]). Different settings for STDGP and GSGP deserve to be studied in the future.

Regarding SLIM, the inflate and deflate operators of each variant were used as two independent mutations with probabilities tuned for each problem [28]. The mutation step was always random, but within an interval that is also problem-specific [28]. For the Toxicity dataset, all the variants used inflate/deflate probabilities of 0.1/0.9 and the $[0, 0.1]$ interval for the random mutation step. For the Concrete dataset, the inflate/deflate probabilities were 0.5/0.5 and the interval of the mutation step was $[0, 3]$. For the remaining problems, the inflate/deflate probabilities were 0.3/0.7 and the interval of the mutation step was $[0, 1]$.

12.6.2 Experimental Results

Figure 12.6 contains the results obtained on the Toxicity, Istanbul and Energy test problems. The leftmost column shows the evolution of the RMSE on the training set; the middle column shows the evolution of the RMSE on the test set; the rightmost

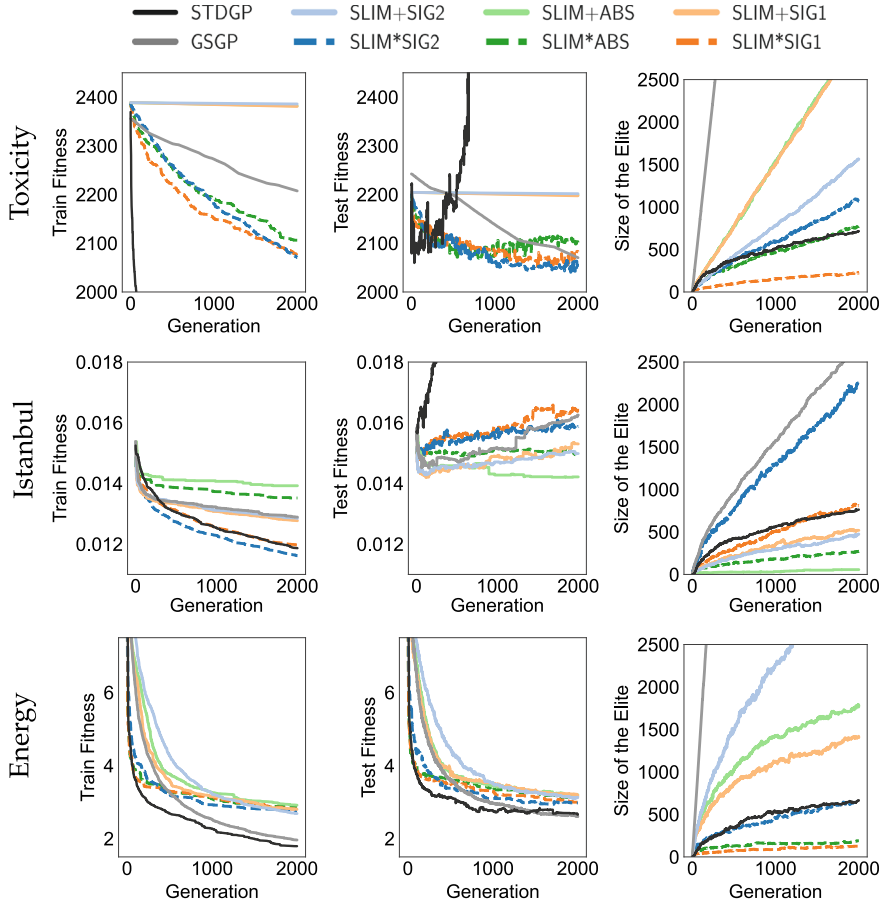


Fig. 12.6 Evolution of the RMSE on training and test, and of the size of the elite on the Toxicity, Istanbul and Energy datasets

column shows the evolution of the size, calculated as the number of tree nodes of the best individual in the population. All the shown values are medians of the 30 runs. The analogous plots for the PPB, Concrete and RBSP problems are shown in Fig. 12.7. We alert for an inconsistency found in the plots presented in [28] where, except for the Concrete problem, one of the lines (there identified as SLIM+1, here renamed SLIM+ABS) was actually representing a preliminary study of SLIM*SIG1. To verify the statistical significance of the test RMSE differences between the SLIM

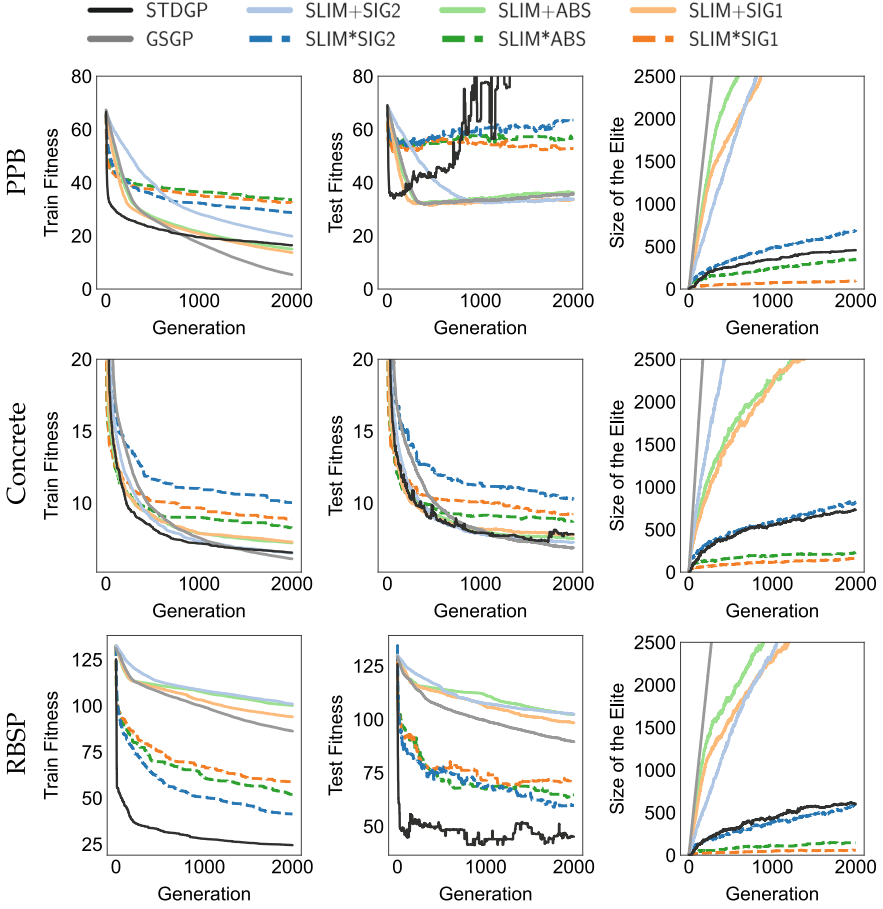


Fig. 12.7 Evolution of the RMSE on training and test, and of the size of the elite on the PPB, Concrete and RBSP datasets

variants and both STDGP and GSGP, Table 12.1 reports the p -values returned by the Wilcoxon rank-sum test with a significance level $\alpha = 0.05$ with the null hypothesis that the samples do not have equal medians of the RMSE on the test set. Analogously, Table 12.2 reports the p -values of the differences in model size.

Toxicity. Looking at Fig. 12.6, on the Toxicity dataset we immediately observe that STDGP strongly overfits. Another observation is that all the SLIM+ variants exhibit similar behavior to each other, with very little evolution from the beginning to the end of the runs, all showing overlapped lines that are nearly horizontal (but not completely), both on training and test. This underfitting behavior was already visible in [28], suggesting that the problem-specific parameters adopted there (and also used here) may be improved by incorporating also method-specific choices. Also, we observe that SLIM* improves both training and test RMSE much faster than GSGP,

Table 12.1 *P*-values returned by the Wilcoxon rank-sum test for the *test RMSE* at termination. Values in bold indicate a statistically significant difference. A checkmark following the *p*-value means that the SLIM variant is better

Toxicity		Istanbul	Energy	PPB	Concrete	RBSP
SLIM+SIG2	STDGP	7.99×10^{-6} ✓	0.13	9.22×10^{-6} ✓	0.03 ✓	4.18×10^{-4}
	GSGP	9.90×10^{-5}	1.42×10^{-6}	0.02 ✓	0.20	7.30×10^{-4}
SLIM*SIG2	STDGP	1.63×10^{-7} ✓	0.07	0.01 ✓	0.27	0.10
	GSGP	0.63	3.45×10^{-5}	1.86×10^{-9}	4.42×10^{-6}	2.09×10^{-4} ✓
SLIM+ABS	STDGP	7.99×10^{-6} ✓	0.04	1.06×10^{-5} ✓	0.09	2.83×10^{-4}
	GSGP	1.53×10^{-4}	1.30×10^{-8}	0.58	0.05	2.56×10^{-3}
SLIM*ABS	STDGP	2.37×10^{-5} ✓	0.10	0.04 ✓	0.78	0.27
	GSGP	0.37	3.74×10^{-3} ✓	1.86×10^{-9}	2.37×10^{-3}	5.14×10^{-6} ✓
SLIM+SIG1	STDGP	7.99×10^{-6} ✓	0.13	6.91×10^{-6} ✓	0.25	9.51×10^{-4}
	GSGP	1.70×10^{-4}	1.42×10^{-6}	0.04 ✓	0.01	0.02
SLIM*SIG1	STDGP	7.91×10^{-5} ✓	0.11	1.58×10^{-3} ✓	0.53	9.93×10^{-3}
	GSGP	0.38	4.42×10^{-6}	1.86×10^{-9}	1.68×10^{-6}	0.01 ✓

although GSGP catches up on test RMSE at the end of the run, when SLIM*ABS is already overfitting and both SLIM*SIG2 and SLIM*SIG1 are threatening to. Looking at Table 12.1, we verify that none of the SLIM* variants show statistically significant differences from GSGP. However, large differences are observed when we look at the evolution of model size, where GSGP produces by far the largest models. Looking at Table 12.2, we verify that SLIM models are *always* significantly smaller than the ones of GSGP, the reason why any further statistical comparisons of size will be focused only on the differences between SLIM and STDGP. Although quite pointless due to the extreme STDGP overfitting, if we compare STDGP and SLIM model sizes we realize that SLIM*SIG1 produces models significantly smaller than STDGP, followed by SLIM*ABS whose models have a comparable size to the ones of STDGP.

Istanbul. On the Istanbul problem, STDGP once again clearly overfits. In fact, most other methods also overfit except both ABS variants, with SLIM+ABS being the clear winner on this problem: it fits the training set less, while maintaining a better generalization ability on unseen data than all other methods, and notably also producing the smallest models. Table 12.1 tells us that none of the SLIM variants is outperformed, neither by STDGP nor by GSGP, and all SLIM variants significantly outperform STDGP. Regarding model size, as stated before, GSGP is always the worst method. Given the clear overfitting of STDGP, it is also quite meaningless to compare it with the SLIM methods, but we do notice that only one SLIM variant, SLIM*SIG2, produces models larger than STDGP, while most variants generate significantly smaller models than STDGP.

Energy. Regarding test RMSE on this problem, all the SLIM variants perform comparably to STDGP and are outperformed by GSGP. Regarding model size, all SLIM+ variants are significantly worse than STDGP (a common observation in most problems); among SLIM*, both SLIM*SIG1 and SLIM*ABS generate models significantly smaller than the ones of STDGP, while SLIM*SIG2 generates models of comparable size to STDGP.

PPB. Also on this problem, STDGP overfits. The SLIM* variants do not achieve good test RMSE. Between GSGP and the SLIM+ variants, both SLIM+SIG2 and SLIM+SIG1 generalize significantly better than GSGP, while SLIM+ABS performs comparably to GSGP. Regarding model size, both SLIM*ABS and SLIM*SIG1 generate models significantly smaller than STDGP, even though they do not perform well on this problem.

Concrete. On this problem, GSGP performs better than most, but SLIM+SIG2 and (borderline) SLIM+ABS are able to match it. Regarding STDGP, the SLIM variants match its performance, with SLIM+SIG2 being able to surpass it. In terms of size, we observe nothing different from the previous problem.

RBSP. STDGP is the method with the best median test RMSE on this problem, but the dispersion of values achieved on the different runs is rather high ($IQR > 30$) and therefore both SLIM*SIG2 and SLIM*ABS achieve a statistically similar test RMSE.

Table 12.3 Number of cases in which each SLIM variant returned significantly better or comparable solutions (strictly better between parenthesis) than GSGP and STDGP in terms of test RMSE and size

	SLIM+SIG2	SLIM*SIG2	SLIM+ABS	SLIM*ABS	SLIM+SIG1	SLIM*SIG1
vs GSGP RMSE	3 (2)	3 (1)	3 (1)	3 (2)	2 (1)	3 (1)
vs STDGP RMSE	5 (4)	6 (3)	4 (3)	6 (3)	5 (3)	5 (3)
vs GSGP Size	6 (6)	6 (6)	6 (6)	6 (6)	6 (6)	6 (6)
vs STDGP Size	1 (1)	3 (0)	1 (1)	6 (5)	1 (1)	6 (5)
Total	15 (13)	18 (10)	14 (11)	21 (16)	14 (11)	20 (15)

SLIM*SIG2 is also able to produce models of comparable size to the ones of STDGP. Regarding the other methods, all SLIM* variants outperform GSGP, contrarily to the SLIM+ ones.

Discussion. The previous results do not identify one method that is overall better than the others. However, if we combine the test RMSE results and the ones concerning the size of the final models for all six problems, some interesting considerations can be made. Table 12.3 reports, for each SLIM variant, the number of times (over the six problems) that this variant returned a smaller or comparable test RMSE and a smaller or comparable model size than traditional GSGP and STDGP. This allows for a quick comparison between the different methods.

The first aspect we notice is that, in general, the SLIM* variants are better than the SLIM+ ones. The difference between SLIM* and SLIM+ is particularly visible in what concerns model size. Our interpretation is that SLIM* achieves smaller models because it can perform larger steps in the semantic space at each mutation event, as shown by the scatterplots in Sect. 12.5. Among the SLIM* variants, the ones that use only one expression to obtain a function with outputs in $[-ms, ms]$ seem better than SLIM*SIG2, again due to their smaller model sizes. We can therefore state that, even though all SLIM variants look promising and worth further investigation, SLIM*ABS and SLIM*SIG1 are the ones that have returned the best results in our set of studied problems. Both match or outperform GSGP in test RMSE on half of the problems while always generating smaller models; both match or outperform STDGP in all problems (except once for SLIM*SIG1) while always generating models that are smaller or of comparable size.

12.7 Conclusions and Future Work

This chapter has extended and deepened a recent publication introducing the Semantic Learning algorithm based on Inflate and deflate Mutation (SLIM_GSGP) [28], a novel version of Geometric Semantic Genetic Programming (GSGP) that enables the generation of compact human-readable models. The distinctive feature of SLIM is its

incorporation, alongside the traditional geometric semantic mutation (herein referred to as inflate mutation), of a deflate mutation. Deflate mutation produces offspring smaller in size than their parents, while retaining all characteristics of a geometric semantic mutation, including the ability to define a ball mutation in the semantic space, and consequently inducing a unimodal error surface for any supervised learning problem. In comparison to [28], this chapter has introduced two additional variants of SLIM and delved into a more detailed discussion of the dynamics and geometric characteristics of their respective inflate and deflate mutation operators. The results have demonstrated that SLIM exhibits high competitiveness with traditional GSGP in terms of error on unseen data while being notably advantageous in terms of model size. SLIM also normally outperforms standard GP in its learning and generalization ability, avoiding overfitting on most problems. Furthermore, it often produces models even smaller than the ones of standard GP. Among the SLIM variants studied, we have observed that the ones employing multiplication to define a ball mutation in the semantic space are, on average, advantageous compared to the ones using addition, both in their ability to generate more accurate models on unseen data and in their significantly smaller models. In conclusion, this chapter clearly demonstrates that SLIM, although a new technique with much yet to be explored and understood, shows significant promise.

This study opens promising avenues for extensive future investigation. A thorough examination of the hyperparameters of the different SLIM variants stands as a pivotal next step. Expanding the scope of the fitness function beyond mere employment of RMSE on the training set presents significant promise. Multiobjective optimization, especially regarding the simultaneous optimization of model size and functional complexity, emerges as a compelling direction. Exploring optimal strategies for implementing the deflate mutation offers an intriguing opportunity. This involves a strategic timing for its application and an informed choice of which part of the parent genotype to deflate, instead of the current random choice. Moreover, we are studying novel crossover operators, capable of generating offspring that are smaller than their parents.

Acknowledgements This work was supported by national funds through FCT (Fundação para a Ciência e a Tecnologia), under the project UIDB/04152/2020 (<https://doi.org/10.54499/UIDB/04152/2020>)—Centro de Investigação em Gestão de Informação (MagIC)/NOVA IMS and through the LASIGE R&D Unit (UIDB/00408/2020 (<https://doi.org/10.54499/UIDB/00408/2020>) and UIDP/00408/2020 (<https://doi.org/10.54499/UIDP/00408/2020>)).

References

1. Akbilgic, O., Bozdogan, H., Balaban, M.E.: A novel hybrid rbf neural networks model as a forecaster. *Stat. Comput.* **24**(3), 365–375 (2014). <https://doi.org/10.1007/s11222-013-9375-7>
2. Archetti, F., Lanzeni, S., Messina, E., Vanneschi, L.: Genetic programming for computational pharmacokinetics in drug discovery and development. *Genet. Program Evolvable Mach.* **8**(4),

- 413–432 (2007)
3. Bakurov, I., Muñoz Contreras, J.M., Castelli, M., Rodrigues, N., Silva, S., Trujillo, L., Vanneschi, L.: Geometric semantic genetic programming with normalized and standardized random programs. *Genet. Program. Evol. Mach.* **25**(1) (2024). <https://doi.org/10.1007/s10710-024-09479-1>
 4. Castelli, M., Manzoni, L.: GSGP-C++ 2.0: A geometric semantic genetic programming framework. *SoftwareX* **10**, 100,313 (2019)
 5. Castelli, M., Manzoni, L., Gonçalves, I., Vanneschi, L., Trujillo, L., Silva, S.: An analysis of geometric semantic crossover: a computational geometry approach. In: *International Joint Conference on Computational Intelligence* (2016)
 6. Castelli, M., Manzoni, L., Mariot, L., Menara, G., Pietropolli, G.: The effect of multi-generational selection in geometric semantic genetic programming. *Appl. Sci.* **12**(10), 4836 (2022)
 7. Castelli, M., Trujillo, L., Vanneschi, L.: Energy consumption forecasting using semantic-based genetic programming with local search optimizer. *Intell. Neurosci.* **2015** (2015). <https://doi.org/10.1155/2015/971908>
 8. Castelli, M., Trujillo, L., Vanneschi, L., Silva, S., Z-Flores, E., Legrand, P.: Geometric semantic genetic programming with local search. In: *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO '15*, p. 999–1006. Association for Computing Machinery, New York (2015). <https://doi.org/10.1145/2739480.2754795>
 9. Castelli, M., Vanneschi, L., Popovič, A.: Controlling individuals growth in semantic genetic programming through elitist replacement. *Intell. Neurosci.* **2016** (2016). <https://doi.org/10.1155/2016/8326760>
 10. Castelli, M., Vanneschi, L., Silva, S.: Prediction of high performance concrete strength using genetic programming with geometric semantic genetic operators. *Expert Syst. Appl.* **40**(17), 6856–6862 (2013)
 11. Dubitzky, W., Granzow, M., Berrar, D.P.: *Fundamentals of Data Mining in Genomics and Proteomics*. Springer (2006)
 12. Farinati, D., Bakurov, I., Vanneschi, L.: A study of dynamic populations in geometric semantic genetic programming. *Inf. Sci.* **648**, 119,513 (2023). <https://doi.org/10.1016/j.ins.2023.119513>. <https://www.sciencedirect.com/science/article/pii/S0020025523010988>
 13. Gonçalves, I., Silva, S., Fonseca, C.M.: On the generalization ability of geometric semantic genetic programming. In: Machado, P., Heywood, M.I., McDermott, J., Castelli, M., García-Sánchez, P., Burelli, P., Risi, S., Sim, K. (eds.) *Genetic Programming*, pp. 41–52. Springer International Publishing, Cham (2015)
 14. Koza, J.R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA (1992)
 15. Martins, J.F.B.S., Oliveira, L.O.V.B., Miranda, L.F., Casadei, F., Pappa, G.L.: Solving the exponential growth of symbolic regression trees in geometric semantic genetic programming. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '18*, p. 1151–1158. Association for Computing Machinery, New York, NY, USA (2018). <https://doi.org/10.1145/3205455.3205593>
 16. McDermott, J., Agapitos, A., Brabazon, A., O'Neill, M.: Geometric semantic genetic programming for financial data. In: *Applications of Evolutionary Computation: 17th European Conference, EvoApplications 2014, Granada, Spain, April 23–25, 2014, Revised Selected Papers 17*, pp. 215–226. Springer (2014)
 17. Moraglio, A.: An efficient implementation of gsgp using higher-order functions and memoization. In: Johnson, C., Krawiec, K., Moraglio, A., O'Neill, M., (eds.), *Semantic Methods in Genetic Programming*. Ljubljana, Slovenia (2014). <http://www.cs.put.poznan.pl/kkrawiec/smgp2014/uploads/Site/Moraglio2.pdf>. Workshop at Parallel Problem Solving from Nature 2014 conference
 18. Moraglio, A., Krawiec, K., Johnson, C.G.: Geometric semantic genetic programming. In: Coello Coello, C.A., Cutello, V., Deb, K., Forrest, S., Nicosia, G., Pavone, M., (eds.) *Parallel Problem Solving from Nature, LNCS*, vol. 7491, pp. 21–31. Springer (2012)

19. Moraglio, A., Mambrini, A.: Runtime analysis of mutation-based geometric semantic genetic programming for basis functions regression. In: E.A. *et al.* (ed.) Proceedings of the annual international conference on Genetic and Evolutionary Computation, GECCO '13, pp. 989–996. ACM, New York, NY, USA (2013)
20. Pawlak, T.P., Krawiec, K.: Competent geometric semantic genetic programming for symbolic regression and boolean function synthesis. *Evol. Comput.* **26**(2), 177–212 (2018)
21. Pietropolli, G., Camerota Verdù, F.J., Manzoni, L., Castelli, M.: Parametrizing gp trees for better symbolic regression performance through gradient descent. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation, pp. 619–622 (2023)
22. Pietropolli, G., Manzoni, L., Paoletti, A., Castelli, M.: Combining geometric semantic gp with gradient-descent optimization. In: Medvet, E., Pappa, G., Xue, B. (eds.) *Genetic Programming*, pp. 19–33. Springer International Publishing, Cham (2022)
23. Poli, R., Langdon, W.B., McPhee, N.F.: *A field guide to genetic programming* (2008)
24. Rafiei, M.: Residential Building Data Set. UCI Machine Learning Repository (2018). <https://doi.org/10.24432/C5S896>
25. Silva, S., Vanneschi, L.: Operator equalisation, bloat and overfitting: a study on human oral bioavailability prediction. In: Rothlauf, F. (ed.) *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8–12, 2009*, pp. 1115–1122. ACM (2009)
26. Tsanas, A., Xifara, A.: Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools. *Energy Build.* **49**, 560–567 (2012). <https://doi.org/10.1016/j.enbuild.2012.03.003>. <https://www.sciencedirect.com/science/article/pii/S037877881200151X>
27. Vanneschi, L.: *An Introduction to Geometric Semantic Genetic Programming*, pp. 3–42. Springer International Publishing, Cham (2017)
28. Vanneschi, L.: SLIM_GSGP: The non-bloating geometric semantic genetic programming. In: Giacobini, M., Xue, B., Manzoni, L. (eds.) *Genetic Programming*, pp. 125–141. Springer Nature Switzerland, Cham (2024)
29. Vanneschi, L., Castelli, M., Gonçalves, I., Manzoni, L., Silva, S.: Geometric semantic genetic programming for biomedical applications: a state of the art upgrade. In: *2017 IEEE Congress on Evolutionary Computation (CEC)*, pp. 177–184 (2017). <https://doi.org/10.1109/CEC.2017.7969311>
30. Vanneschi, L., Castelli, M., Manzoni, L., Silva, S.: A new implementation of geometric semantic GP and its application to problems in pharmacokinetics. In: Krawiec, K., Moraglio, A., Hu, T., Uyar, A.S., Hu, B., (eds.) *Proceedings of EuroGP, LNCS*, pp. 205–216. Springer (2013)
31. Vanneschi, L., Castelli, M., Scott, K., Trujillo, L.: Alignment-based genetic programming for real life applications. *Swarm Evolut. Comput.* **44**, 840–851 (2019). <https://doi.org/10.1016/j.swevo.2018.09.006>. <https://www.sciencedirect.com/science/article/pii/S2210650218300208>
32. Vanneschi, L., Silva, S.: *Lectures on Intelligent Systems. Natural Computing Series*. Springer (2023). <https://doi.org/10.1007/978-3-031-17922-8>
33. Vanneschi, L., Silva, S., Castelli, M., Manzoni, L.: Geometric semantic genetic programming for real life applications. In: Riolo, R., Vladislavleva, K., Moore, J. (eds.) *Genetic Programming Theory and Practice XI. Genetic and Evolutionary Computation. Computer Science Collection*. Springer, US (2013)

Chapter 13

Revisiting Gradient-Based Local Search in Symbolic Regression



Bogdan Burlacu, Stephan M. Winkler, and Michael Affenzeller

Abstract Gradient descent-based local search can dramatically improve solution performance in symbolic regression tasks, at the cost of significantly higher runtime as well as increased risks of overfitting. In this paper, we investigate exactly what amount of local search is really needed within the GP population. We show that low intensity local search is sufficient to boost the fitness of the entire population, provided that local search information in the form of optimized numerical parameters is written back into the genotype at least some of the time. Our results suggest that spontaneous adaptations (in the Lamarckian sense) act as evolutionary fuel for the Baldwin effect in genetic programming, and that in the absence of the former, the latter does not occur and evolution is hindered. The Lamarckian model works particularly well in symbolic regression, as local search only affects model coefficients and does not affect the inheritance of useful building blocks contained in the model structure.

13.1 Introduction

Symbolic Regression (SR) augmented with gradient-based optimization of model coefficients was shown to be a top performer in a variety of regression tasks [5, 14]. This type of local search hybrid is generally known as memetic search or memetic genetic programming [22]. Here, while the evolutionary search meta-heuristic controls the global search strategy and guides the population toward optima in the fitness landscape, local search is assimilated with learning at the level of the individual

B. Burlacu (✉) · M. Affenzeller

Department of Software Engineering, University of Applied Sciences Upper Austria, Hagenberg, Austria

e-mail: bogdan.burlacu@fh-ooe.at

M. Affenzeller

e-mail: michael.affenzeller@fh-ooe.at

S. M. Winkler

Department of Medical and Bioinformatics, University of Applied Sciences Upper Austria, Hagenberg, Austria

e-mail: stephan.winkler@fh-ooe.at

and plays a support role preventing premature convergence and in general, improving the algorithm's ability to maintain an optimal balance between exploration and exploitation. The specific manner of learning plays an important role in the evolutionary process. Two learning models are of particular importance in evolutionary algorithms [13, 26]:

1. *Lamarckian learning model*. This model is based on the model of evolution proposed by the French botanist Jean-Baptiste Lamarck, who suggested that individuals adapt to their environment during their lifetime and then pass these adaptations to their offspring. Although disproved in biology, this model is possible within the framework of memetic algorithms, if local search is allowed to directly update the individual genotypes.
2. *Baldwinian learning model*. Proposed by the American psychologist James Mark Baldwin, this model suggests that the ability to learn new behaviors will improve an individual's reproductive success. Therefore, it is sufficient that an individual's fitness is improved by local search without the need to update its genotype.

In both cases, investing resources into local search may substantially increase the computational requirements of the algorithm. In this work, we investigate the overhead of local search and establish a baseline minimum computational requirement for local search to remain effective. Additionally, we look at its impact on population dynamics and algorithm convergence.

13.2 Local Search in Symbolic Regression

Symbolic regression evolves tree data structures that encode mathematical expressions, where internal nodes encode functions and terminal nodes encode dataset variables or free parameters. The set of function symbols $\mathcal{F}$ together with the set of terminal symbols $\mathcal{T}$ form the primitive set $\mathcal{P} = \mathcal{F} \cup \mathcal{T}$ of the algorithm. When $\mathcal{F}$ contains only differentiable functions, then the gradient of a tree model can be easily obtained through automatic differentiation.

Several gradient-based local search approaches have been proposed in the literature. A number of these methods employ line search [25], while others use trust region methods [4, 18, 19, 29] and some use backpropagation [15]. Some methods employ numerical differentiation [18, 29] while most employ automatic differentiation [4, 6, 12, 15, 17, 19, 25, 31]. The choice of optimizer also has a big effect on the quality of the outcome. de Melo et al. [10] find that in least-squares optimization scenarios typical for SR, the Levenberg-Marquardt algorithm [21] is both the fastest and the most performant in terms of solution quality and that derivative-free methods are generally much slower than gradient-based methods. A summary of gradient-based local search approaches is given in Table 13.1.

Table 13.1 Summary of gradient descent-based local search approaches in genetic programming

Method/Author(s)	AD mode	Description
Topchy and Punch [25]	Reverse	• Symbolic regression (synthetic benchmarks) • Parameters: 100 individuals, $3 \cdot 10^4$ evaluations • LS applied to the entire population • Fixed learning rate $\eta = 0.5$, three steps of gradient descent
Zhang and Zmart [31]	Reverse	• Symbolic classification (synthetic benchmarks) • Parameters: 300/500 individuals, 51 generations • LS applied to the top 5% individuals in the population • Both online and offline gradient descent tested • Fixed learning rate chosen from $\eta \in [0.2, 0.4, 0.7, 1.0, 1.4]$
Graff et al. [15]	Reverse	• Symbolic regression (wind power time series data) • Params: 10 individuals, 10^5 generations, max. length 1000 • LS applied only on the best individual. • Uses <i>resilient propagation</i> (RPROP) for gradient descent
Chen et al. [6]	Reverse	• Symbolic regression (synthetic benchmarks) • Parameters: 200 individuals, 50 generations • Fixed learning rate $\eta = 0.5$, up to three iterations
Izzo et al. [17]	Forward	• Symbolic regression (exact expression discovery) • Params: $(1 + \lambda)$ evolution model, $\lambda = 4$, 2000 generations • High order AD via truncated Taylor polynomials • Can solve differential equations and find prime integrals • Combines line search ($\eta = 0.05$) with batch learning
Cranmer [8, 9]	Forward	• Symbolic regression • Equation discovery
Dick et al. [12]	Reverse	• Symbolic regression (synthetic benchmarks) • Parameters [12]: 100 individuals, 100 generations • Parameters [20]: 200, 500 individuals, 5000 generations • Uses <i>stochastic gradient descent</i> (SGD) • Fixed learning rate $\eta = 0.5$
Operon Burlacu et al. [4]	Forward	• Symbolic regression (synthetic and real-world data) • Parameters: $5 \cdot 10^5$ [5], 10^6 [20] evaluations, grid search • LS applied to the entire population • Up to ten gradient descent steps • Adaptive learning rate (trust region)
LS-OSGP Kommenda et al. [19]	Reverse	• Symbolic regression (synthetic and real-world data) • Parameters: grid search with maximum 10^5 evaluations • LS applied to the entire population • Adaptive learning rate (trust region), ten iterations
LSALL-GP Z-Flores et al. [29] Juarez-Smith et al. [18]	Numerical	• Symbolic regression (synthetic and real-world data) • Parameters [29]: 500 individuals, $2 \cdot 10^6$ evaluations • Parameters [18]: 100 individuals, 10^5 evaluations • LS applied to all, best, worst individual(s), or probabilistic
DGP [30]	Reverse	• Symbolic regression

The performance of local search in Symbolic Regression critically depends on the accurate and efficient computation of function derivatives. Here, reverse-mode automatic differentiation, which can be implemented efficiently for symbolic expression

trees, plays an important role. A brief description of this process and its application to gradient descent optimization is given in the following sections.

13.2.1 Automatic Differentiation

We present here a very succinct introduction to automatic (algorithmic) differentiation (AD), as applied to symbolic expression trees. For a thorough treatment of the topic, we turn the reader to more established literature [2, 16]. AD is based on the idea that any numerical computation can be described as a composition of a finite set of elementary operations for which derivatives are known. The derivative of said computation is then obtained by aggregating the derivatives of the constituent operations through the chain rule of calculus. Given a target function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, the Jacobian matrix J is the matrix of all first-order partial derivatives of f :

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}_{m \times n} \quad (13.1)$$

This matrix captures the rate of change of each component of f with respect to each component of input variable $x \in \mathbb{R}^n$. If f is a composite function $f(x) = g(h(x))$, $x \in \mathbb{R}^n$, $g : \mathbb{R}^k \rightarrow \mathbb{R}^m$, $h : \mathbb{R}^n \rightarrow \mathbb{R}^k$ then, according to the chain rule, the (i, j) -th element of the Jacobian is

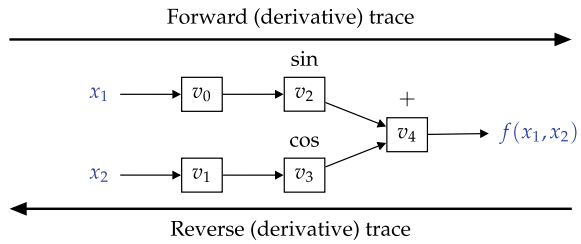
$$J_{ij} = \frac{\partial f_i}{\partial x_j} = \frac{\partial g_i}{\partial h_1} \frac{\partial h_1}{\partial x_j} + \frac{\partial g_i}{\partial h_2} \frac{\partial h_2}{\partial x_j} + \dots + \frac{\partial g_i}{\partial h_k} \frac{\partial h_k}{\partial x_j} \quad (13.2)$$

The chain rule can be applied to arbitrarily deep function compositions. In order to compute the Jacobian, AD requires all the intermediate quantities resulting from the evaluation of the composite elementary operations to be stored. This list of intermediate results is called a *primal trace* or alternatively a Wengert list [27].

In what follows, we will use the concept of a computational graph [1] (a directed acyclic graph of function compositions) to illustrate the sequence of elementary operations being carried out and reason about the mechanism of AD. Depending on the direction in which the computational graph is traversed, we distinguish between:

Forward-mode AD This mode (also called *tangent mode*) numerically evaluates the action of the Jacobian matrix from inputs to outputs (that is to say, derivatives of the *inner* functions are substituted repeatedly). It is often considered the more simple and “natural” form of AD due to the fact that the order of evaluation coincides with the flow of derivative information; therefore, all the quantities involved can be computed at the same time and no additional memory is required. Forward AD performs as many passes over the computational graph equal as the

Fig. 13.1 Computational graph of the function $f(x_1, x_2) = \sin(x_1) + \cos(x_2)$



number of function inputs. In practice, *dual numbers* [7] are typically used to implement forward-mode AD.

Reverse-mode AD This mode (also called *adjoint* or *cotangent linear* mode) numerically evaluates the action of the Jacobian matrix from outputs to inputs (the derivatives of the *outer* functions are substituted repeatedly). The derivative of a chosen dependent variable with respect to an intermediate variable v_k is called an *adjoint* and represents the sensitivity of f_j with respect to changes in v_k . Reverse AD is a second phase process: in the first phase, a forward pass evaluates the original function and stores all intermediate variables. Then, in the second phase derivatives are calculated by propagating adjoints in reverse, from outputs to inputs.

Thus, reverse AD performs as many passes over the computational graph as the number of function outputs. Since tree individuals in SR typically encode single-output functions, reverse-mode brings a significant runtime advantage as all the partial derivatives can be computed in a single pass. Moreover, since SR individuals are trees (as opposed to more general directed acyclic graph structures), the implementation of reverse-mode in SR can be simplified.

An example computational graph for the function $f(x_1, x_2) = \sin(x_1) + \cos(x_2)$ is illustrated in Fig. 13.1. Using reverse-mode, its derivatives can be computed in a single reverse AD pass as shown in Table 13.2.

13.2.2 Gradient Descent

We consider the problem of finding the optimal numerical values for model coefficients using the gradient of the model function. An SR model represented by a symbolic expression tree can be described as the function

$$\hat{f}(\mathbf{x}, \theta) : \mathbb{R}^{m \times p} \rightarrow \mathbb{R}$$

where $\mathbf{x} \in \mathbb{R}^m$ is a single observation of m feature values and $\theta \in \mathbb{R}^p$ is a vector of coefficients associated with the tree nodes. The optimization problem can be framed as a minimization over parameter space $\Theta \subset \mathbb{R}^p$

Table 13.2 Reverse-mode AD of $f(x_1, x_2) = \sin(x_1) + \cos(x_2)$ at $(x_1, x_2) = (2, 3)$ **Primal trace**

$$\begin{aligned}
v_0 &= x_1 &&= 2 \\
v_1 &= x_2 &&= 3 \\
v_2 &= \sin(v_0) = \sin(2) &&= 0.909 \\
v_3 &= \cos(v_1) = \cos(3) &&= -0.990 \\
v_4 &= v_2 + v_3 = 0.909 - 0.990 &&= -0.081 \\
\mathbf{f} &= \mathbf{v}_4 &&= -0.081
\end{aligned}$$

Reverse adjoint (derivative) trace

$$\begin{aligned}
\bar{v}_4 &= \bar{f} &&= 1 \\
\bar{v}_3 &= \bar{v}_4 \partial v_4 / \partial v_3 = 1 \cdot 1 &&= 1 \\
\bar{v}_2 &= \bar{v}_4 \partial v_4 / \partial v_2 = 1 \cdot 1 &&= 1 \\
\bar{v}_1 &= \bar{v}_3 \partial v_3 / \partial v_1 = 1 \cdot -\sin(3) &&= -0.141 \\
\bar{v}_0 &= \bar{v}_2 \partial v_2 / \partial v_0 = 1 \cdot \cos(2) &&= -0.416 \\
\bar{\mathbf{x}}_2 &= \bar{\mathbf{v}}_1 &&= -0.141 &&= \partial \mathbf{f} / \partial \mathbf{x}_2 \\
\bar{\mathbf{x}}_1 &= \bar{\mathbf{v}}_0 &&= -0.416 &&= \partial \mathbf{f} / \partial \mathbf{x}_1
\end{aligned}$$

$$\operatorname{argmin}_{\theta \in \Theta} \|y - \hat{f}(\mathbf{x}, \theta)\|_2^2$$

For any fixed input $\mathbf{x}$ and corresponding target value y , let $\mathcal{L}(\theta) = \|y - \hat{f}(\mathbf{x}, \theta)\|_2^2$ be a loss function with gradient $\nabla \mathcal{L}(\theta) \in \mathbb{R}^p$. The necessary condition for θ^* to be a local minimum point (or a local maximum point, or an inflection point) is that

$$\nabla \mathcal{L}(\theta^*) = 0$$

which leads to

$$\nabla \mathcal{L}(\theta) = \nabla_{\hat{f}}^{\top}(\mathbf{x}; \theta) \mathcal{L}(\theta) = 0$$

Then, we can use

$$\mathcal{L}(\theta + \Delta\theta) \approx \mathcal{L}(\theta) + \nabla(\theta) \Delta\theta$$

and solve the least-squares problem:

$$\min_{\Delta\theta} = \frac{1}{2} \|\nabla(\theta) \Delta\theta + \mathcal{L}(\theta)\|^2$$

Convergence depends on appropriate tuning of the step size $\Delta\theta$, for which multiple mathematical techniques exist, such as trust region or line search methods [23].

13.3 Methods

In this study we make use of the Operon library [4] which already includes local search functionality with gradient descent supported by reverse-mode automatic

Table 13.3 Benchmark problems

Id	Name	Features	Instances	Training range	Test range
F_1	Airfoil Self Noise [3]	5	1503	[0, 1000)	[1000, 1503)
F_2	Poly-10 [24]	10	500	[0, 250)	[250, 500)
F_3	Concrete [28]	8	1000	[0, 500)	[500, 1000)

Table 13.4 Algorithm configuration

Parameter	Value
Population size	1000 individuals
Maximum generations	1000 generations
First objective	R^2 coefficient of determination (with linear scaling)
Second objective	Tree length
Evaluation budget	10^7 evaluations
Maximum tree length	50 nodes
Maximum tree depth	10 levels
Crossover probability	100%
Mutation probability	25%
Mutation types	Point mutation, change function, change variable replace subtree, remove subtree, insert subtree
Selection scheme	Tournament selector (tournament size = 5)
Local search parameters	Gradient descent (Levenberg-Marquardt algorithm)
Gradient descent iterations n_{iter}	Varied by grid search between [0, 5] in increments of 1
Local search probability p_{LS}	Varied by grid search between [0, 1] in increments of 0.1
Lamarckian probability p_{LM}	Varied by grid search between [0, 1] in increments of 0.1

differentiation. We have modified the library to allow configurable probabilities for local search as follows:

- Local search probability p_{LS} is the probability of applying local search
- Lamarckian probability p_{LM} is the probability of updating the genotype

We perform empirical tests using the NSGA-II [11] algorithm on the benchmark problems detailed in Table 13.3. The configuration of the algorithm is detailed in Table 13.4.

We note that without local search, evolving a population of 1000 individuals over 1000 generations requires exactly 10^6 solution evaluations. The difference up to the

total budget of 10^7 evaluations was added in order to allow sufficient budget for the local search iterations in which model residuals and Jacobians have to be computed.

13.4 Results

In order to understand the effects of local search intensity, as well as the relationship between Lamarckian evolution and the Baldwin effect, we perform a grid search over the parameter space defined by the three local search-related parameters: n_{iter} , p_{LS} , and p_{LM} . For each problem and parameter configuration, we perform 50 repetitions and aggregate the results, obtaining $11 \times 11 \times 6 \times 50 = 36\,300$ runs per problem. The experiments are performed on a Ryzen™ 3950X processor running at 4 Ghz.

13.4.1 Local Search Intensity and Solution Performance

Local search intensity can be controlled via the two parameters: the number of gradient descent iterations n_{iter} and the local search probability p_{LS} . We use color-coding with the help of heatmaps to show the results in a more concise manner in Fig. 13.2. The full results are available online.¹ Despite some variability in the results, we note that GP without local search (when either n_{iter} or p_{LS} are zero) performs worse than using any amount of local search. This remains true even when allowed to run for 10 000 generations, in order to fully exhaust the evaluation budget. Concerning the local search aspect, a purely Baldwinian evolution model ($p_{\text{LM}} = 0$) hinders evolution, leading to poor solutions, as indicated by the top row in each of the heatmaps.

Overall, looking at the quality values of the final solutions returned by the algorithm, we find no evidence to suggest that a higher number of local search iterations might be beneficial on the tested problems.

The results strongly indicate that, due to the combined effects of Lamarckism and inheritance, one single iteration of the gradient descent procedure is sufficient to improve performance. We note that low quality models obtained without any form of Lamarckism are not caused by overfitting (training R^2 values are similarly low),

¹ <https://github.com/foolnotion/gptp-workshop-2024>.

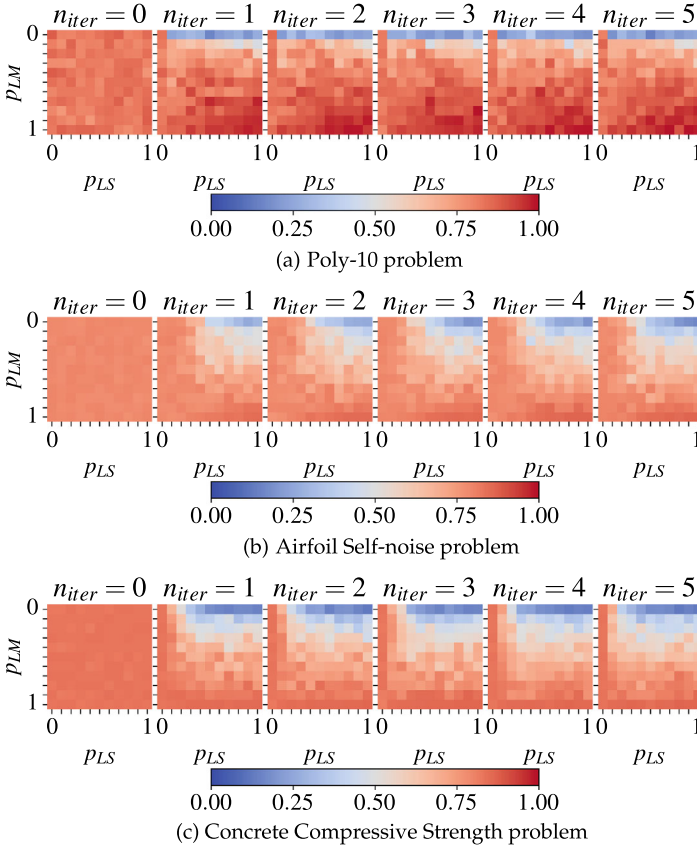


Fig. 13.2 Heatmap visualization of mean test R^2 values over the parameter grid. GP without local search (when either n_{iter} or p_{LS} are zero) performs worse than using any amount of local search

but rather seem to be caused by the bias introduced by optimized fitness values in the selection scheme, which does not truly reflect the evolvability of the population, thus confusing the selection process. Furthermore, in the face of destructive crossover operations, learning potential cannot be inherited.

13.4.2 Local Search Effects on Convergence

For this analysis, we fix $p_{LS} = 1$ and $p_{LM} = 1$, looking at the influence of gradient descent iterations on the mean quality curves shown in Fig. 13.3. These curves are computed using R^2 value averages over a discretized timeline divided into 10ms intervals, over the duration of

- the fastest standard GP run (out of fifty), for the top row of charts;
- the fastest run of each respective configuration (out of fifty), for the bottom row.

We use the top row convergence curves to estimate convergence speed as the area under the curve (AUC) using a simple integration method (Simpson's 1/3 rule).

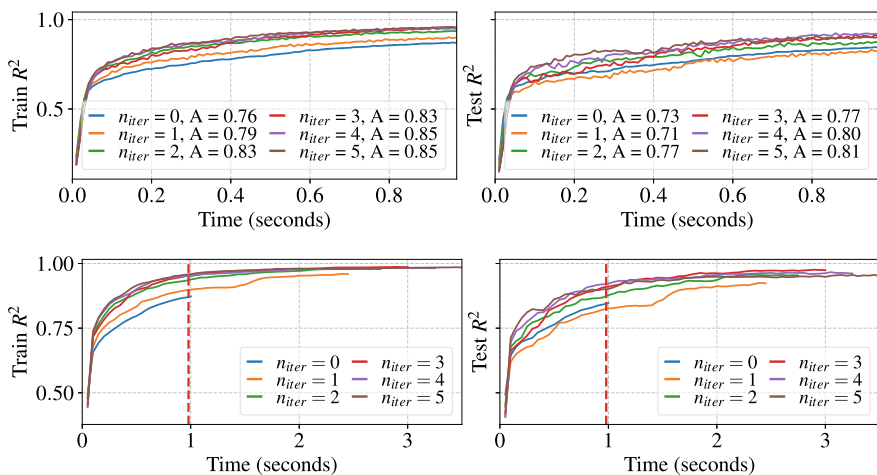
Based on these results, we can conclude that local search tends to improve convergence speed, even when accounting for the additional computational effort and runtime. To further analyze this aspect and to show that a large evaluation budget is better spent on local search than on increasing run length, we perform an additional comparison with GP runs (without local search) that were allowed to continue over 10 000 generations. We note that this leads to overfitting on two out of the three tested problems (*Poly-10* and *Airfoil*), while still being outperformed in terms of training quality.

13.4.3 Local Search Effects on Model Length

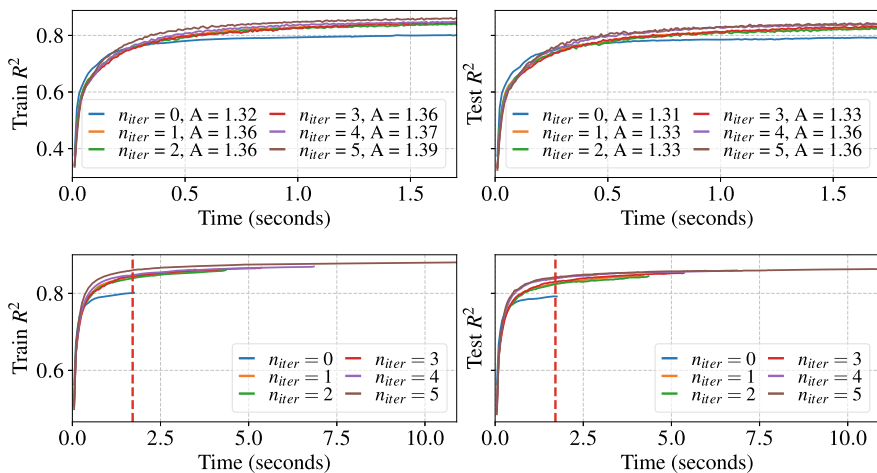
Somewhat surprisingly and contrary to previous results (e.g. [25]), we observe that while local search runs are usually characterized by higher average model length, the best solutions can potentially be significantly smaller than baseline GP as shown by Table 13.3a for the *Poly-10* problem. The explanation for this is a combined effect of using model length as a secondary objective in the NSGA-II algorithm and local search affording individuals a certain degree of robustness against deleterious recombination events, thus reducing the need for large structures that provide more buffering and protection. However, this effect is not observed on the *Airfoil* and *Concrete* problems, where both average and best solution length are larger with local search. In general, the tendency toward increased model length with local search is problem-dependent (Fig. 13.4).

13.4.4 Local Search Effects on Runtime

While it is clear that local search increases runtime, it is worth checking if significant differences occur, e.g. due to the different impact of each local search configuration on the evolutionary dynamics. However, we observe no significant impact of Lamarckian probability p_{LM} on runtime. Local search probability p_{LS} impacts runtime proportionally, while the biggest factor is the number of gradient descent iterations, n_{iter} . Runtime variation with n_{iter} is shown in Table 13.5.



(a) Poly-10 problem



(b) Airfoil problem

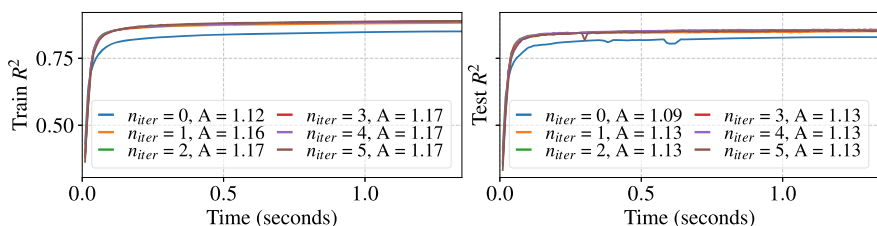
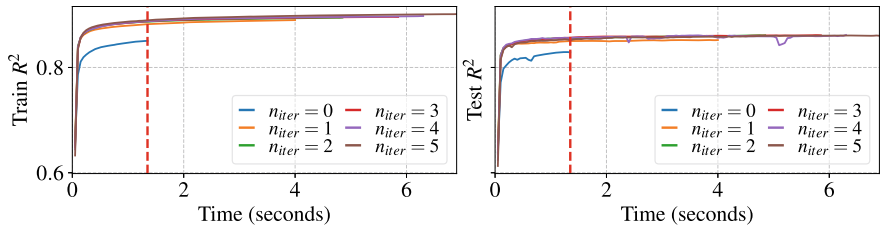
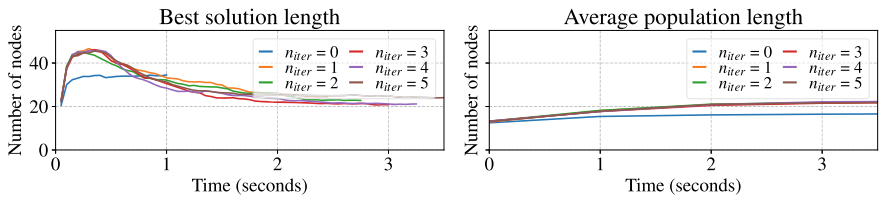


Fig. 13.3 Algorithm convergence speed, over the duration of the fastest GP run without local search (top row of charts) and with local search (bottom row of charts)

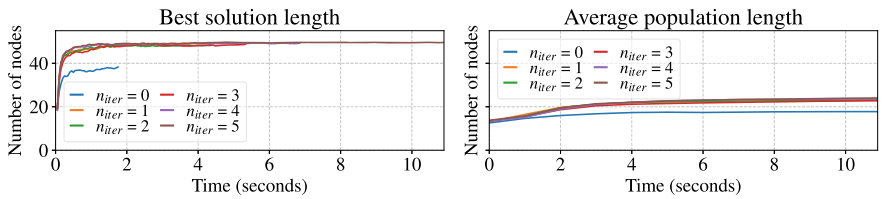


(c) Concrete problem

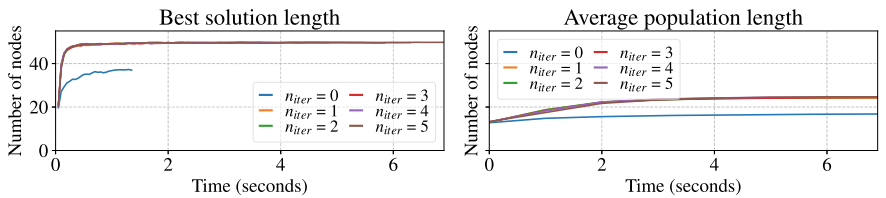
Fig. 13.3 (continued)



(a) Poly-10 problem



(b) Airfoil Self-noise problem



(c) Concrete Compressive Strength problem

Fig. 13.4 Evolution of best and average solution lengths

Table 13.5 Algorithm runtime (measured in seconds)

n_{iter}	0	1	2	3	4	5
Poly-10	1.23	3.06	3.55	3.84	4.13	4.6
Airfoil	1.85	6.56	8.19	9.78	11.09	12.66
Concrete	1.69	4.32	5.30	6.33	7.32	7.92

13.5 Discussion

In this work we have investigated the effects of local search intensity in symbolic regression. We have performed an exhaustive grid search over the parameter space defined by local search probability, Lamarckian probability (defined as the probability that learned model coefficients are written back to the genotype), and gradient descent iterations. Our results have shown that despite being more computationally intensive, local search increases speed of convergence (defined as quality improvement over elapsed runtime) and therefore can potentially produce higher quality results *faster* than standard GP. More importantly, a low amount of local search (e.g. local search probability around 50%, using one gradient descent iteration) seems to work just as well as larger amounts, suggesting that a synergy between Lamarckism and inheritance can compensate for reduced local search effort as local search information is preserved across generations. Concerning the Baldwin effect in symbolic regression, our results indicate that without any form of Lamarckian evolution, the Baldwin effect alone is unable to sustain evolvability, as it introduces a disconnect between fitness and the inheritance of useful traits and building blocks, hindering the evolutionary search. More experiments will be required to elucidate this phenomenon in more detail. Future research will investigate other approaches to local search, for instance replacing the Levenberg-Marquardt algorithm with Stochastic Gradient Descent in order to further improve computational efficiency.

Acknowledgements The authors gratefully acknowledge the reviewers' comments which helped to improve the manuscript.

References

1. Bauer, F.L.: Computational graphs and rounding error. *SIAM J. Numer. Anal.* **11**(1), 87–96 (1974). <https://doi.org/10.1137/0711010>
2. Baydin, A.G., Pearlmutter, B.A., Radul, A.A., Siskind, J.M.: Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.* **18**(1), 5595–5637 (2017)
3. Brooks, T.F., Pope, D.S., Marcolini, M.A.: Airfoil self-noise and prediction. Tech. Rep. 1218, NASA (1989)
4. Burlacu, B., Kronberger, G., Kommenda, M.: Operon C++: an efficient genetic programming framework for symbolic regression. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, GECCO '20*, pp. 1562–1570. Association for Computing Machinery, internet (2020). <https://doi.org/10.1145/3377929.3398099>
5. Cava, W.G.L., Orzechowski, P., Burlacu, B., de França, F.O., Virgolin, M., Jin, Y., Kommenda, M., Moore, J.H.: Contemporary symbolic regression methods and their relative performance. *CoRR* [arXiv:abs/2107.14351](https://arxiv.org/abs/2107.14351) (2021)
6. Chen, Q., Xue, B., Zhang, M.: Generalisation and domain adaptation in gp with gradient descent for symbolic regression. In: *2015 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1137–1144 (2015). <https://doi.org/10.1109/CEC.2015.7257017>
7. Clifford: Preliminary sketch of biquaternions. In: *Proceedings of The London Mathematical Society*, pp. 381–395 (1871)

8. Cranmer, M.: Pysr: Fast and parallelized symbolic regression in python/julia (2020). <https://doi.org/10.5281/zenodo.4041459>
9. Cranmer, M.: Interpretable machine learning for science with pysr and symbolicregression.jl (2023)
10. de Melo, V.V., Fowler, B., Banzhaf, W.: Evaluating methods for constant optimization of symbolic regression benchmark problems. In: 2015 Brazilian Conference on Intelligent Systems (BRACIS), pp. 25–30. Natal, Brazil (2015). <https://doi.org/10.1109/BRACIS.2015.55>
11. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Trans. Evol. Comput.* **6**(2), 182–197 (2002). <https://doi.org/10.1109/4235.996017>
12. Dick, G., Owen, C.A., Whigham, P.A.: Feature standardisation and coefficient optimisation for effective symbolic regression. In: Coello Coello, C.A., et al. (eds.), *Proceedings of the 2020 Genetic and Evolutionary Computation Conference, GECCO '20*, pp. 306–314. Association for Computing Machinery, internet (2020). <https://doi.org/10.1145/3377930.3390237>
13. El-Mihoub, T.A., Hopgood, A.A., Nolle, L., Battersby, A.: Hybrid genetic algorithms: a review. *Eng. Lett.* **13**(2), 124–137 (2006)
14. de Franca, F.O., Virgolin, M., Kommenda, M., Majumder, M.S., Cranmer, M., Espada, G., Ingelse, L., Fonseca, A., Landajuela, M., Petersen, B., Glatt, R., Mundhenk, N., Lee, C.S., Hochhalter, J.D., Randall, D.L., Kamienny, P., Zhang, H., Dick, G., Simon, A., Burlacu, B., Kasak, J., Machado, M., Wilstrup, C., Cava, W.G.L.: Interpretable symbolic regression for data science: analysis of the 2022 competition (2023)
15. Graff, M., Pena, R., Medina, A.: Wind speed forecasting using genetic programming. In: de la Fraga, L.G., (ed.) *2013 IEEE Conference on Evolutionary Computation*, vol. 1, pp. 408–415. Cancun, Mexico (2013). <https://doi.org/10.1109/CEC.2013.6557598>
16. Griewank, A., Walther, A.: *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd edn. Society for Industrial and Applied Mathematics, USA (2008)
17. Izzo, D., Biscani, F., Mereta, A.: Differentiable genetic programming. In: *Genetic Programming: 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19–21, 2017, Proceedings 20*, pp. 35–51. Springer (2017)
18. Juarez-Smith, P., Trujillo, L.: Integrating local search within neat-GP. In: *GECCO '16 Companion: Proceedings of the Companion Publication of the 2016 Annual Conference on Genetic and Evolutionary Computation*, pp. 993–996. ACM, Denver (2016). <https://doi.org/10.1145/2908961.2931659>
19. Kommenda, M., Burlacu, B., Kronberger, G., Affenzeller, M.: Parameter identification for symbolic regression using nonlinear least squares. *Genet. Program. Evolvable Mach.* **21**(3), 471–501 (2020). <https://doi.org/10.1007/s10710-019-09371-3>. <https://rdcu.be/cAIuP>. Special Issue on Integrating numerical optimization methods with genetic programming
20. La Cava, W.: Results files from the 2022 srbench competition: interpretable symbolic regression for data science (2022). <https://doi.org/10.5281/zenodo.6842176>
21. Moré, J.J.: The levenberg-marquardt algorithm: implementation and theory. In: Watson, G.A. (ed.) *Numerical Analysis*, pp. 105–116. Springer, Berlin (1978)
22. Moscato, P.: *On evolution, search, optimization, genetic algorithms and martial arts: towards memetic algorithms*. Tech. Rep. C3P Report 826, California Institute of Technology (1989)
23. Nocedal, J., Wright, S.: *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, New York (2006). <https://books.google.at/books?id=VbHYoSylFcC>
24. Poli, R.: A simple but theoretically-motivated method to control bloat in genetic programming. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., Costa, E. (eds.) *Genetic Programming*, pp. 204–217. Springer, Berlin (2003)
25. Topchy, A., Punch, W.F.: Faster genetic programming based on local gradient search of numeric leaf values. In: *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation, GECCO'01*, pp. 155–162. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)

26. Turney, P.D.: Myths and legends of the baldwin effect. CoRR cs.LG/0212036 (2002). <http://arxiv.org/abs/cs/0212036>
27. Wengert, R.E.: A simple automatic derivative evaluation program. Commun. ACM **7**(8), 463–464 (1964). <https://doi.org/10.1145/355586.364791>
28. Yeh, I.C.: Modeling of strength of high-performance concrete using artificial neural networks. Cem. Concr. Res. **28**(12), 1797–1808 (1998)
29. Z-Flores, E., Trujillo, L., Schuetze, O., Legrand, P.: Evaluating the effects of local search in genetic programming. In: Tantar, A.A., Tantar, E., Sun, J.Q., Zhang, W., Ding, Q., Schuetze, O., Emmerich, M., Legrand, P., Del Moral, P., Coello Coello, C.A., (eds.) EVOLVE - A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation V. Advances in Intelligent Systems and Computing, vol. 288, pp. 213–228. Springer, Peking (2014). https://doi.org/10.1007/978-3-319-07494-8_15. <https://hal.inria.fr/hal-01060315>
30. Zeng, P., Song, X., Lensen, A., Ou, Y., Sun, Y., Zhang, M., Lv, J.: Differentiable genetic programming for high-dimensional symbolic regression (2023)
31. Zhang, M., Smart, W.: Genetic programming with gradient descent search for multiclass object classification. In: Keijzer, M., O'Reilly, U.M., Lucas, S.M., Costa, E., Soule, T., (eds.) Genetic Programming 7th European Conference, EuroGP 2004, Proceedings, LNCS, vol. 3003, pp. 399–408. Springer, Coimbra (2004). https://doi.org/10.1007/978-3-540-24650-3_38

Chapter 14

It's Time to Revisit the Use of FPGAs for Genetic Programming



Christopher Crary, Greg Stitt, Bogdan Burlacu, and Wolfgang Banzhaf

Abstract In the past, field-programmable gate arrays (FPGAs) have had some notable successes when employed for Boolean and fixed-point genetic programming (GP) systems, but the more common floating-point representations were largely off limits, due to a general lack of efficient device support. However, recent work suggests that for both the training and inference phases of floating-point-based GP, contemporary FPGA technologies may enable significant performance and energy improvements—potentially multiple orders of magnitude—when compared to general-purpose CPU/GPU devices. In this chapter, we highlight the potential advantages and challenges of using FPGAs for GP systems, and we showcase how novel algorithmic considerations likely need to be made in order to extract the most benefits from specialized hardware. Primarily, we consider tree-based GP, although we include suggestions for other program representations. Overall, we conclude that the GP community should earnestly revisit the use of FPGA devices, especially the tailoring of state-of-the-art algorithms to FPGAs, since valuable enhancements may be realized. Most notably, FPGAs may allow for faster and/or less costly GP runs, in which case it may also be possible for better solutions to be found when allowing an FPGA to consume the same amount of runtime/energy as another platform.

C. Crary (✉) · G. Stitt

Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA
e-mail: ccrary@ufl.edu

G. Stitt

e-mail: gstitt@ufl.edu

B. Burlacu

Heuristic and Evolutionary Algorithms Laboratory, University of Applied Sciences Upper Austria, Hagenberg, Upper Austria, Austria
e-mail: bogdan.burlacu@fh-ooe.at

W. Banzhaf

Department of Computer Science and Engineering, Michigan State University, East Lansing, MI, USA
e-mail: banzhafw@msu.edu

14.1 Introduction

Recent trends in machine learning highlight the need for energy-efficient computation, during both training and inference [25, 69]. For example, despite widespread success, neural networks often consume prohibitive amounts of power and energy for many use cases [8, 69], in addition to posing considerable scaling challenges for well-established use cases, such as data centers [1, 8, 69]. Such limitations can motivate other learning systems, such as genetic programming (GP) [7, 41, 55], where it has been widely shown that the pairing of evolutionary search with alternative model structures (e.g., trees, assembly languages, tangled program graphs, etc.) can sometimes allow for more compact solutions and enhanced efficiency during inference [39, 44, 55]. However, training often remains complex with current GP techniques, which motivates improvements to training efficiency [11, 17, 55].

There are various ways to improve the training efficiency of GP, and they generally involve either increasing performance (i.e., throughput) or enhancing energy efficiency, for which there are at least four key benefits: (1) with increased performance, useful solutions can potentially be found in a shorter amount of time; (2) with improved energy efficiency, there is the potential for lower operational costs, which (3) can allow for more cost-effective multi-computer GP systems, in turn allowing for higher performance; and (4) with either improved performance or improved energy efficiency, better solutions can potentially be found when allowing the system to consume a similar amount of runtime/energy as before.

In regard to its computational model, GP is in a remarkable yet challenging position. Although the algorithms of GP are often embarrassingly parallel [55], which can open the door to extremely high-performance and energy-efficient computation, the most widely accessible computing platforms—CPUs and GPUs—are not quite built for the multiple-program, multiple-data model of GP. For CPUs, the use of multiple cores/threads is relatively straightforward, but it is often difficult or prohibitively expensive (in terms of power and other costs) to continually scale up [30]. And for GPUs, which offer numerous simpler cores and have been highly successful in accelerating other forms of machine learning, the need for conditional program execution (e.g., to decide which function primitive to execute) and large cache sizes generally limits acceleration capabilities [13, 17, 58].

Ideally, we would have a computing platform that more perfectly aligns with the computational model of GP, in order to improve performance and energy efficiency. In general, achieving such an alignment is the motivation for *domain-specific architectures*, for which either an application-specific integrated circuit (ASIC) or a reconfigurable computing platform such as a field-programmable gate array (FPGA) can be leveraged [30]. Although designing specialized hardware via an ASIC allows for the most flexibility in achieving desired performance and power/energy characteristics [30], FPGAs allow for reconfigurability and lower development costs, which we believe lowers the barrier to entry for any initial research efforts within the GP community. Ultimately, successful FPGA designs can be repurposed for an ASIC, with the goal of obtaining additional performance and energy benefits.

In this chapter, we establish various potential benefits and challenges of implementing modern GP with modern FPGAs. We loosely base our work on the state-of-the-art Operon tool [10, 11, 44], which is floating-point-based and tree-based, but we also include considerations for other GP domains. In years past, the application of FPGAs to floating-point-based GP was largely off limits, due to older technologies that did not efficiently support floating-point [31]. Now, with the advent of floating-point multiply-adders, along with the potential for numerous other resources and higher clock frequencies, improvements of multiple orders of magnitude may be accessible with FPGAs [16, 17]. Nevertheless, efficient mappings of GP to specialized hardware remain challenging, and we believe that various algorithmic considerations should be made in order to maximize amenability, by which we generally mean that unnecessary complexity should be stripped out.

For example, it is suggested in [17] that a key optimization needed for achieving significant computational efficiency with an FPGA relies on the idea of executing GP function primitives with a minimal amount of shared hardware resources. Unfortunately, designing such a resource-sharing mechanism for standard implementations of floating-point operators is challenging, due to high-performance requirements of the relevant algorithms often leading to overly complex or obfuscated realizations. However, this begs an important question: *does GP need standard operators?* For floating-point domains, could more efficient implementations that more roughly approximate the corresponding continuous operators suffice for evolution? In [16], we establish that the answer is *yes*, such operators can suffice.

As another example, we plan to integrate local search techniques [55], but we foresee that more complex algorithms such as Levenberg-Marquardt may not be appropriate for a hardware accelerator, due to a memory complexity that could likely create a performance bottleneck [40]. A more appropriate choice for local search may be a form of batched gradient descent, but additional studies are likely needed to determine effects on solution quality, especially in conjunction with other algorithmic simplifications. Also, similar considerations should be made for multi-objective evolutionary algorithms, such as NSGA-II [55].

Beyond efforts to accelerate the training procedures of GP with FPGAs, which is the main focus of this chapter, we note that the use of FPGAs for deploying final models is also of considerable interest. For instance, although a physical tree-based architecture may not be the best solution for accelerating the evaluation of arbitrary tree-based programs (Sect. 14.4), implementing a final tree-based model with such an architecture could minimize the latency and runtime of individual predictions, which is likely desirable. Ultimately, the reconfigurable nature of FPGAs readily allows for separate specialized architectures during inference, which could enable GP to provide even more competitive machine learning solutions [18].

The remainder of this chapter is as follows. In Sect. 14.2, we list some related work. In Sect. 14.3, we provide some relevant background on modern computing and FPGA devices. In Sect. 14.4, we detail some instances of how FPGA-based hardware architectures could be devised for GP. In Sect. 14.5, we establish various challenges and motivate future work. In Sect. 14.6, we conclude our study.

14.2 Related Work

Since prior work on FPGA-based GP is not frequently referenced, we provide a brief listing of all such works of which we are aware. Similar to how comparing different GP representations (e.g., tree-based GP and linear GP) can be tricky, comparing specialized hardware architectures—especially across GP domains—is problematic. In addition, with most of the following works employing significantly outdated FPGA devices, the advantages/disadvantages of each system are not clear within the context of modern technologies. Ultimately, due to these issues as well as space constraints, we largely avoid establishing either comparisons or results.

For traditional tree-based GP [41, 55], we refer the reader to [17, 23, 42, 62]. For Linear GP [9], see [12, 21, 32, 50]. For Cartesian GP [52], see [19, 61, 72, 73]. For geometric semantic GP [53], see [27, 49]. For grammar-guided GP [51], stack-based GP [65], tangled program graphs [39], and other well-known GP representations, we are unaware of implementations leveraging FPGA devices. Separately, we note that FPGAs have also been used for the application area of *evolvable hardware* [75, 76], although this has been primarily for evolving circuit-based solutions, rather than for improving the performance/energy characteristics of the GP procedure itself.

Of the aforementioned FPGA-based systems, it appears that the systems given in [12, 17, 23, 24] utilized floating-point, where [17] was our work on an initial tree-based accelerator, and where [23, 24] independently proposed an architecture conceptually similar to ours, with ours providing several important contributions, as described in [17]. The architecture presented in [12] was for linear GP, and it is a significant departure from the other floating-point-based architectures listed.

14.3 Background

In this section, we provide some relevant background on modern (digital) computing, which should motivate the exploration of FPGA-based architectures for GP.

14.3.1 A Brief Overview of Modern Computing

For about twenty years now, the practical relevance of *Moore's Law* has been waning, and *Dennard scaling* no longer applies [30]. In brief, Moore's Law describes an empirical regularity that the maximum number of transistors in an integrated circuit chip doubles roughly every two years,¹ and Dennard scaling refers to the idea that as transistor circuit area scales down, power density roughly stays the same.

¹ Recent work allows us to conclude that Moore's Law is still alive [59].

Notably, from the mid-1980s to the early-2000s, a combination of Moore's Law and Dennard scaling allowed average CPU performance to roughly double (i.e., execution times to roughly halve) every two years [30]. Then, from the end of Dennard scaling in the early-2000s until the late-2010s, the use of multiple *general-purpose* cores per chip kept Moore's Law alive, but various theoretical and practical barriers led to a significant slowing of performance enhancements. Primarily, performance enhancements were constrained by consistent power budgets—which, in general, are constrained by electromigration, mechanical, and thermal limits—as well as the limits on parallelism as prescribed by *Amdahl's Law* [6, 30]. With consistent power budgets due to physical constraints, the number of general-purpose cores in a single chip approached a practical upper limit [22, 30]. Thus entered the next big trend, which is still relevant today: *domain-specific architectures* (DSAs). Importantly, with DSAs, hardware specialized to a particular application domain can often accomplish more with a similar, sometimes smaller, power budget [30, 54, 56, 67, 70].

In general, domain-specific architectures can offer equivalent, and sometimes better, performance and energy benefits when compared to modern general-purpose architectures, such as central processing units (CPUs) and graphics processing units (GPUs) [30, 54, 56, 67, 70]. Although DSAs can sometimes serve as complete solutions, the latest trend is to integrate both general-purpose and domain-specific *chipllets* into a single circuit, so that the system can efficiently support a wide range of applications while additionally being optimized for a particular subset [30]. Such a composite system is often referred to as a *system-on-chip* (SoC). One recent, notable example of an SoC is the Apple M1 Ultra chip, which consists of 114 billion transistors primarily allocated to a 20-core CPU, 64-core GPU, 32-core “Neural Engine,” and high-bandwidth memory. In terms of number of transistors, this chip represents roughly a *50 million times increase* from the Intel 4004 chip—the first commercially produced microprocessor—which consisted of 2,300 transistors.

14.3.2 Domain-Specific Architectures

Many application domains can benefit from the use of a domain-specific architecture (DSA) [30, 43]. In general, a DSA can be leveraged when either (1) large amounts of algorithmic parallelism can be exploited or (2) some low-power, low-area, or specialized implementation is desired. However, practically speaking, additional factors must often be considered, such as those involving nonrecurring engineering (NRE) time, NRE cost, unit cost, sale volume, and sale price [30, 66].

Naturally, there are different mechanisms for designing a DSA, and each comes with its own set of trade-offs. Generally speaking, an *application-specific integrated circuit* (ASIC) provides the most flexibility in achieving desired performance and power/energy characteristics, as well as low unit costs, but this route usually requires years of NRE time and millions of dollars in NRE costs [30]. Unfortunately, according to *Rock's Law* [20], NRE costs often increase significantly with newer device technologies, which continually makes ASIC engineering a technical and economic

challenge.² Thus, ASICs are usually most applicable for applications in which large NRE times/costs are tolerable and high sale volumes/prices are expected, so that NRE cost may be offset by a large amount of low-cost, high-profit sales.

Besides the fabrication of an ASIC, another alternative for designing a DSA is to utilize a *reconfigurable computing (RC)* system [30, 71]. In essence, RC systems are programmable computing systems in which specialized digital circuitry can be synthesized from different levels of abstraction, without recourse to integrated circuit development.³ Oftentimes, RC systems gain appeal by trading off higher unit costs for both (1) lower NRE times/costs and (2) comparable performance/energy benefits. In general, RC systems are useful for prototyping designs before ASIC development or for developing standalone solutions in which either (1) NRE cost must be low or (2) high unit cost can be amortized by high sale volume/price or the ability to reconfigure the device over time. Additionally, with the ability to design high-performance, energy-efficient solutions, and the ability to support different hardware designs over multiple reconfigurations, RC platforms are used for various research [54, 56, 60, 67, 70]. The most popular type of RC system is a *field-programmable gate array (FPGA)*, which we detail in the next subsection.

14.3.3 Field-Programmable Gate Arrays

Unfortunately, the name *field-programmable gate array* fails to capture the main mechanism by which FPGAs implement designs—there, in fact, does not exist any array of (logic) gates [29]. Primarily, there exist many small memories, known as *lookup tables (LUTs)*, which can implement the truth table(s) corresponding to some desired circuitry. For a simple example of such an implementation, see Fig 14.1. By way of LUTs, FPGAs support combinational logic. To additionally support sequential logic, FPGAs also increasingly leverage *flip-flop* memory components [29]. Ultimately, combinations of LUT and flip-flop components can allow for highly flexible circuit configurations, but implementing a system only with such components may not be feasible, depending on design complexity. To address this fact, modern FPGA systems also contain more coarse-grained resources, such as integer and floating-point multiply-adders,⁴ high-bandwidth memories, general-purpose CPU cores, and sometimes other specialized computing cores (e.g., “AI engines” [2]), so that common computing tasks can more readily be implemented [2, 34, 54, 66, 70]. In this respect, modern FPGAs are, themselves, SoC devices. Overall, by integrating many thousands or millions of components via a reconfigurable interconnect, FPGAs can

² However, the manufacturing of older technologies generally becomes cheaper over time [30].

³ An RC system is itself an ASIC, yet an ASIC exposing some aspect(s) of reconfigurability.

⁴ These multiply-adders are often referred to as *digital signal processing (DSP) blocks/engines*, or just *DSPs* for simplicity. For floating-point, newer devices can efficiently implement single/half precision and “bfloat16” [4, 14, 26, 35], although double precision is still relatively complex.

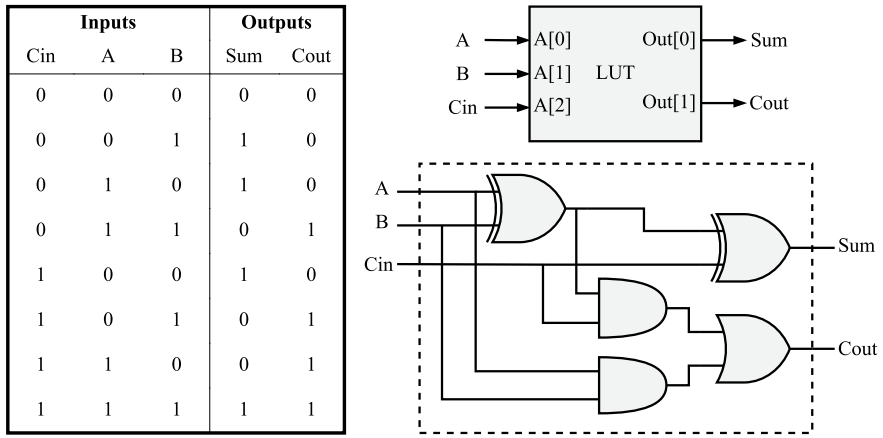


Fig. 14.1 A LUT-based implementation of a full adder, with carry-in and carry-out. Importantly, the depicted LUT memory could implement not only this circuit, but *all* 3-input, 2-output digital circuits. Also, note that the schematic in the bottom-right is just for illustration; with a LUT-based implementation, no logic gates are used. For a more thorough example, see [29, Example 5.5]

implement massively parallel designs, many of which provide significant performance/energy benefits when compared to CPU/GPU systems [30, 60].

Importantly, the aforementioned characteristics suggest that modern FPGAs may be an attractive implementation option for genetic programming (GP), where algorithms are often embarrassingly parallel and often exhibit both data-level and function-level parallelism. In the next section, we illustrate this concept further.

14.4 Applying FPGAs to Genetic Programming

In this section, we identify how modern FPGAs may provide valuable practical enhancements to genetic programming (GP). Most notably, we show how the performance (i.e., throughput) of *floating-point-based* GP may be improved by multiple orders of magnitude when compared to state-of-the-art CPU/GPU systems. We make our case for a form of tree-based GP loosely inspired by Operon [11]—a state-of-the-art tool in terms of both technology and solution quality [10, 44]—but we also provide considerations for other GP representations. We establish performance benefits within the context of program evaluation, since this is generally the performance bottleneck of GP systems [11, 55], and we then consider how evolution could align with such enhancements. Ultimately, designing high-performance hardware for evolution is a major next step for future work. We mainly consider performance rather than power/energy, since the former is more well studied [16, 17], but we note that it has been widely demonstrated that FPGAs provide significant power/energy improvements for various applications [2, 30, 54, 56, 67, 70].

Therefore, if we allow for the possibility of any of such benefits, we can estimate that improvements in *performance-per-watt* measures may be even more significant. Overall, our preliminary results motivate additional studies into FPGA-based GP.

14.4.1 Evaluation

First, we examine the architecture that we previously presented in [17]. As depicted in Fig. 14.2, the architecture leverages a specialized full tree of generic computing resources in order to compute any program relevant to a GP primitive set, as long as the depth of the program is not larger than the depth of the tree, the latter of which is defined by the user. The main motivation behind this architecture is that, with tree-based GP, every program expression is a tree. Therefore, if the architecture provides a physical tree of generic resources such that each resource is capable of computing all function primitives, then the tree can compute *entire programs* in parallel. However, even more notably, if we additionally design the generic resources (i.e., function units) to be *pipelined*, then the architecture can generate an output for an entire program *every clock cycle* after some initial latency, as shown in Fig. 14.2. Although pipelining the tree precludes the existence of arbitrary control structures, such structures are usually unnecessary [10, 44], and pipelining enables data-level and/or function-level parallelism. Lastly, to further increase throughput, our architecture includes a compiler in hardware that translates compact prefix-based expressions into machine codes for the tree while it is evaluating, so that the tree may switch

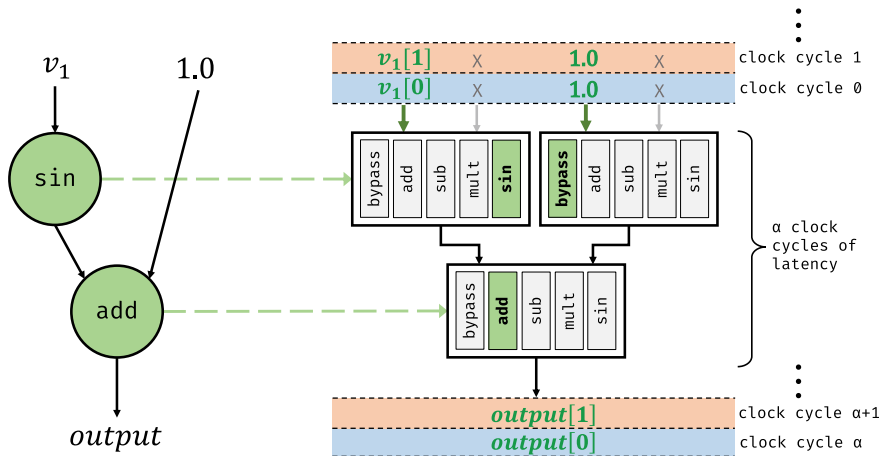


Fig. 14.2 A portrayal of our initial tree-based GP accelerator presented in [17], which can parallelize the evaluation of different data points and different solutions *every clock cycle* via a reconfigurable tree pipeline. Each node of the pipeline can perform any function within the GP primitive set, as well as a bypass, which allows for arbitrary program shapes

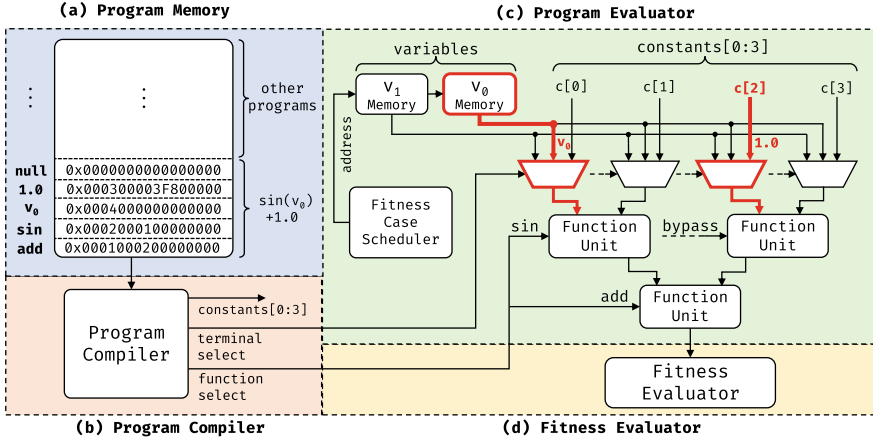


Fig. 14.3 High-level overview of the architecture presented in [17]. Programs (e.g., $\sin(v_0) + 1.0$) are stored in (a) *program memory* and dynamically compiled by the (b) *program compiler* into machine codes for the (c) *program evaluator*. The program evaluator uses a reconfigurable function tree pipeline to execute a compiled expression for a set of fitness cases, resulting in a set of estimated outputs to which the (d) *fitness evaluator* compares a set of desired outputs

between programs *within a single clock cycle*. Importantly, generating outputs for entire programs every clock cycle and changing programs within a single cycle are forms of parallelism that have not been achieved with general-purpose CPU/GPU architectures. For an overall flow of this initial architecture, see Fig. 14.3.

In [17], we compared the aforementioned architecture to several recent CPU/GPU systems using the performance measure *node evaluations per second*, which is effectively the conventional *GP operations per second (GPop/s)*, but without a notion of evolution [13]. Using three different floating-point primitive sets of varying complexity, we showcased that our architecture implemented on a mid-range 14nm FPGA achieved an average speedup of $43\times$ when compared to a recent GPU solution, TensorGP, implemented on 8nm process-node technology, and an average speedup of $4,902\times$ when compared to a popular GP software tool, DEAP, running parallelized across all cores of a 2-socket, 28-core (56-thread), 14nm CPU server. Despite our single-FPGA accelerator being $2.4\times$ slower on average when compared to the state-of-the-art Operon tool executing on the same 2-processor CPU system, our system was the fastest in several instances, and we detailed five potential extensions that could provide a $32\text{--}144\times$ speedup over the initial design. Below, we summarize these extensions, and we include two additional extensions that augment the possible speedup range to be $128\text{--}576\times$. But first, in order to effectively explain the extensions, we highlight the three main challenges of our initial architecture:

1. *Exponential Growth.* For an m -ary tree with $m > 1$, where m is the maximum function arity of the primitive set, the amount of function units needed to implement the tree grows exponentially with increasing tree depth.

2. *Function Unit Complexity.* For the tree architecture to be able to support arbitrary programs, every function unit must support *all* function primitives.
3. *Low Resource Utilization.* For $|F|$ function primitives, the utilization of each function unit in terms of these high-level primitives is at most $\frac{1}{|F|}$, and likely worse since programs are often not full trees. In addition, the utilization of low-level device primitives (e.g., DSPs) can be significantly less.

Now, we summarize the potential extensions which, if all were achieved simultaneously, could allow for a $128\text{--}576\times$ speedup over the initial design:

1. *Use Compacted Trees.* To be able to more effectively leverage device resources as well as support larger program depths/sizes, we could employ “compacted tree” architectures that allow for the use of all resources that are currently unused due to exponential growth. See below for details. (Up to $2\times$ speedup.)
2. *Multiplex Function Unit Resources.* Function unit primitives experience poor utilization due to the fact that they are implemented with independent IP blocks. This issue could be improved upon by implementing a function unit via a single IP block that multiplexes a minimal amount of some device resource(s), e.g., DSPs. Such an “overlay” could free up a significant amount of resources, allowing for further parallelization of program evaluation. (Between $2\text{--}6\times$ speedup.)
3. *Design for Higher Clock Frequencies.* For our accelerator, performance (i.e., throughput) is directly proportional to clock frequency. With modern FPGAs, it is not uncommon for designs to achieve clock frequencies in the range $400\text{--}850$ MHz after optimizing for timing [54, 67, 70]. We estimate that we can achieve up to a $2\text{--}3\times$ higher average clock frequency once we optimize for timing and potentially move to a newer device [17]. (Between $2\text{--}3\times$ speedup.)
4. *Use a Higher-End FPGA Device.* With a more modern, higher-end FPGA implemented on a newer process-node technology (e.g., [34]), we should be able to support at least $1.4\times$ more ALM resources, $2\times$ more DSP resources, and $1.5\times$ more embedded memory resources, all in addition to higher clock frequencies [14]. Separately, such newer devices allow for an additional $2\times$ more DSP resources when using half-precision floating-point or the recent bfloat16 precision [14], which may be useful for GP [11]. (Up to $4\times$ speedup.)
5. *Use Multiple FPGAs.* The CPU results presented in [17] rely on a dual-socket server populated with two CPU packages, whereas we currently only utilize a single FPGA for our accelerator. Therefore, out of fairness, we could parallelize our design across two FPGAs. (Up to $2\times$ speedup.)
6. *Double-buffer GP Runs.* When our accelerator enters the context of a full GP system, including evolution, we aim to execute two GP runs simultaneously, by evolving one population while evaluating another. (Up to $2\times$ speedup.)

In regard to “compacted trees,” we note that we previously suggested two alternative architectures in [17]. Of these two architectures, we are currently most interested in the specialized “linear” processor that would directly execute prefix/postfix representations. Essentially, when compared to our initial architecture, we could potentially achieve better throughput and support larger program sizes/depths by

instantiating many specialized single-unit processing cores in parallel. The execution model of this architecture would be loosely similar to a standard CPU system, but we expect that the number of cores within our architecture could scale better, and we expect that our cores could have higher throughput within the context of GP, since we could execute entire function primitives every clock cycle [17].

In addition to higher throughput and larger program sizes/depths, some other potential benefits of a “specialized CPU” architecture are (1) input bandwidth requirements could be reduced, since multiple cores could share the same inputs and since the evaluation of a single program would take longer—whereas the evaluation of an entire population may take less time when accounting for multiple cores—which may allow datasets to more easily be stored off-chip; (2) the need for program compilation could be removed, since native prefix/postfix representations could be directly interpreted, which would eliminate a possible performance/area bottleneck; and (3) arbitrary control flow could potentially be supported, since every core would effectively only execute one program node at a time. On the other hand, some potential challenges introduced by the specialized CPU architecture are (1) pipelining a sequential execution model could require state memory (e.g., a temporary stack) to be duplicated multiple times, which could limit throughput and program sizes/depths if device resources are exhausted too quickly; and (2) any additional logic that is to be included for evaluation (e.g., fitness calculations, local search, etc.) may have to be included within each core in order to maximize performance, which may create a hardware area bottleneck depending on the complexity of the relevant logic. In any event, further investigations of this architecture are warranted.

For GP representations other than trees, it seems that specialized CPUs may also be attractive. For example, when compared to tree-based GP, it seems plausible that we could similarly pipeline the execution model of linear GP by duplicating register files. Separately, strategies for stack-based GP would likely be similar to tree-based GP, but the existence of stacks for different data types may be challenging for memory consumption. Ultimately, other architectures may be more appropriate for a given GP domain, e.g., spatially parallel architectures may be more appropriate for Cartesian GP [72], but it is still worth considering the use of sequential execution models. Notably, the major semiconductor company AMD has recently embraced similar execution models for neural networks and other domains by embedding specialized CPU cores—“AI engines”—within some of their Versal FPGAs [2].

Besides the architectural extensions already mentioned, we note that other performance enhancements can potentially be extracted from FPGAs, depending on the remaining aspects of the relevant GP system. For example, suppose that local search is to be supported and that a weight and bias term are to be allocated to each program node, similar to Operon [11]. In this context, there effectively can be up to $5\times$ as many nodes in each program once accounting for the extra multiplication and addition operations, and we could potentially compute these extra multiplies in parallel to standard node computations by allocating just one additional DSP resource to each computation core, which would allow for up to another $5\times$ speedup. In addition, when considering more complex evaluation routines, e.g., automatic differentiation and batched gradient descent, other “downstream” computations such

as weight/bias updates could potentially be parallelized as well, which may provide even more flexibility in achieving considerable performance enhancements.

Overall, within the context of tree-based GP, we note that modern FPGAs provide the potential for performance improvements of multiple orders of magnitude when compared to state-of-the-art tree-based CPU/GPU systems, like Operon [11]. In addition, based on various previous studies involving FPGA devices, such performance benefits may enable similarly significant energy improvements [2, 30, 54, 56, 67, 70]. Although it is not yet clear what modern FPGAs may achieve in other GP domains, we note that similar enhancements seem plausible, given that the program interpreter of Operon could be used as a baseline for a generic multiple-program, multiple-data system [11], and given that the hardware resources of an FPGA can be configured in most any manner. At the very least, our current results motivate additional studies into how FPGAs could be applied to various forms of GP.

14.4.2 Evolution

In general, we expect the performance of an overall GP system to largely be governed by how efficient evaluation is implemented, since the runtime of evaluation generally grows with the size of the training dataset, whereas the runtime of evolution often does not [11, 55]. Although Amdahl's Law suggests that evolution could eventually become a bottleneck [30]—since continually optimizing a fixed workload would have diminishing returns—we estimate that the hardware complexity of evolution will often be considerably less than that of evaluation, especially when considering the potential for more complex evaluation routines, like those for gradient-based local search [40]. In addition, if we intend to double-buffer two independent GP runs in order to maximize throughput across GP runs (Sect. 14.4.1), then we could reasonably allow the runtime of evolution to be equal to that of evaluation—since for each generation we could prevent dead cycles for evaluation just by having the evolutionary routines of one GP run be completed by the end of the evaluation routines for the other GP run—which could enable reduced hardware complexity for evolution. Lastly, following Gustafson's Law [28], we may be able to meaningfully augment the GP procedure such that the workload of evaluation is significantly increased without increasing the workload of evolution by the same amount, which may further ease hardware requirements for evolution.

For tree-based GP, we briefly consider possible strategies for tournament selection, subtree crossover, and one-point mutation [55], which are normally employed by the state-of-the-art Operon tool [10, 11, 44].⁵ For tournament selection, hardware should be relatively simple, since we can likely just infer a tree of pipelined comparators in order to find a program that has the minimum/maximum fitness value within some tournament. With a pipelined implementation, tournaments can potentially be decided every clock cycle, which may ultimately mean that only one

⁵ Other operators are often included as well, which we leave for future work (Sect. 14.5.2).

“selection engine” is needed to align with the performance of evaluation. For subtree crossover and one-point mutation, pipelined implementations are likely more challenging, since tree-based programs often need to be parsed in order to determine any viable crossover/mutation point(s) and in order to copy program data. Thus, multiple “variation engines” may be needed in parallel in order to align with potential evaluation speedups, but each engine should be relatively simple, containing mostly just a state machine and some shift registers. For all of these routines, we can likely allocate just a few small buffers for storing random numbers, and then have some independent circuit(s) continually keep the buffers full. For pseudo-random number generation, the *taus88* algorithm seems like a viable candidate, as it is robust and efficient for hardware [48]. Future work can consider other options, if necessary.

For GP representations other than trees, hardware implementations of evolution may be even less complex. For example, some representations readily allow for uniformly random crossover and mutation points, which can reduce or eliminate the need to parse programs during variation, and which in turn may enable the ability to use significantly less hardware when aligning with potential evaluation speedups. This fact motivates additional considerations into how tree-based algorithms could potentially be altered in order to maximize amenability to specialized hardware, and it also motivates continued explorations into which GP representation may be the most appropriate for a given application. Of course, many factors must be considered for the latter. For instance, if gradient-based local search is desired, techniques like automatic differentiation may be fairly straightforward to implement for tree-based expressions, since trees directly encode a “forward” and “reverse” pass, whereas implementations for other representations might be considerably more complex, since a graph structure may first need to be constructed.

Ultimately, it seems plausible that hardware for evolution could align well with the potential performance enhancements laid out for evaluation (Sect. 14.4.1). Therefore, we estimate that modern FPGAs may provide considerable performance/energy improvements to entire GP systems. However, there are still various challenges and unknowns, which we discuss further in the next section.

14.5 Challenges and Future Work

In this section, we discuss some challenges confronting the GP community in regard to the use of FPGA devices, and we then propose various avenues for future work.

14.5.1 Challenges

Although there is considerable potential for performance and power/energy enhancements when employing FPGA devices (Sects. 14.3 and 14.4), current technologies present at least three general challenges [66]:

1. *Productivity*. When compared to standard software development, it is generally regarded that FPGAs have low design productivity. There are multiple reasons for this, but the two main issues are that (1) designing meaningful circuits generally demands considerable *digital design* expertise and (2) device compilation can take hours or even days [66]. To reduce required efforts, many developments have taken place throughout the past few decades, such as the creation of numerous specialized programming languages [37], the extension of pre-existing programming languages (e.g., C++ and Python) [45, 57], the simplification of design tools and their compilation algorithms [63, 74], the creation of overlay circuit technologies [15, 64], and the integration of software-programmable computing cores within FPGA chips [2]. Overall, significant improvements have been made, especially in regard to *high-level synthesis (HLS)* [46], but classical techniques are still often necessary in order to implement a circuit with the desired performance/energy characteristics.⁶ Ultimately, of the various challenges facing the GP community in regard to a wider adoption of FPGA devices, we believe that reduced productivity will likely be the most significant. However, for most of the community, understanding digital design is not required in order to contribute to efforts involving FPGA devices. For example, any work to reduce algorithmic complexities would likely be useful (Sect. 14.5.2).
2. *Amenability*. In general, not all algorithms are directly amenable to FPGA devices [66]. One key reason for this is that various high-level constructs do not have standard mappings to FPGAs, often because such mappings would not be widely useful and/or intuitive [66]. For instance, pointers and recursions do not immediately make sense for most LUT-based circuits (Sect. 14.3.3), and even though support may be possible through additional hardware complexity, alternative constructs are likely more suitable if performance/energy characteristics are important. Separately, even if an algorithm can be directly mapped, it is not guaranteed that an inferred circuit will effectively utilize low-level device resources, and poor mappings often lead to lower computational efficiency. Fortunately, such issues have become less pronounced over time with heterogeneous computing platforms and newer FPGA technologies that support additional and more versatile components [2, 34], but the mapping of certain functionality can still be challenging, and algorithmic tweaks are often necessary. Besides such challenges, another amenability issue is that the complex interconnect within FPGA devices usually necessitates a clock frequency that is significantly lower than modern CPU/GPU devices [66]. For instance, even with high-end technologies, clock frequencies for programmable logic must almost always be set to less than 1 GHz, and generally much lower than that, often being at least a factor of five less than similarly recent CPU/GPU technologies [54, 56, 60, 66, 67, 70]. Thus, when the goal of using an FPGA is to achieve higher performance than CPU/GPU technologies, a massive amount of parallelism is often needed, which is frequently achieved through pipelining and/or pipeline duplication. However, when the goal of using

⁶ For some useful introductions to digital design, see [29, 68].

an FPGA is to leverage enhanced energy efficiency, the lower clock frequencies can often be a plus.

3. *Cost.* Due in part to the aforementioned challenges regarding amenability, there has yet to be a “killer app” for FPGA acceleration [66].⁷ Whereas the development of GPUs was clearly motivated by graphics applications and, now, scientific computing, FPGAs evolved from a considerably smaller market targeting the development of “glue logic” and prototyping for ASIC devices [30, 71]. In general, the larger demand for GPUs has continually led to a significant discrepancy in subsidies for research/development, supply, and device costs. Until recently, it was not uncommon for the latest GPUs to cost a few hundred dollars while the latest FPGAs were listed for at least \$10,000 [66]. Fortunately, this price gap has narrowed and even sometimes inverted, with recent high-end GPU systems costing anywhere from \$1,000–\$40,000 [47], but FPGA devices are still typically pricey or unavailable due to low supply. However, some free or cheap cloud-based platforms with FPGAs now exist; for example, see [3, 5].

14.5.2 Future Work

In the following, we establish several important avenues for future work, and we estimate difficulty with the high-level classifications *easy*, *medium*, and *hard*:

1. *Further explore “specialized CPU” architectures. (Medium to hard.)* As discussed in Sect. 14.4.1, we estimate that simpler architectures specialized to sequential execution models may be generally useful for GP, and especially useful for tree-based GP, as long as both data-level and function-level parallelism can be exploited. In work not yet publicized, we have implemented such an architecture for standard tree-based GP, as well as for a form of tree-based gradient descent, but we have not yet proven scalability. Thus, future work should explore scaling up the number of cores with such an architecture, especially in conjunction with additional complexity, such as high clock frequencies, high-bandwidth memory, and more advanced local search techniques.
2. *Improve amenability of evolutionary algorithms. (Medium to hard.)* In many instances, research ideas that are better aligned with available hardware and software resources have had a greater chance of success [33]. As such, when considering specialized hardware, there is motivation to reconsider whether state-of-the-art evolutionary algorithms are suitable for hardware, and whether any changes should be made. Of course, various trade-offs will inevitably surface if we attempt to redesign algorithms, but there may exist variants of algorithms that allow for better performance/energy characteristics while allowing for comparable (or better) solution quality. Although such work would likely best be accomplished by considering the design of hardware and algorithms in tandem, this need not be

⁷ And if there is a killer app, an ASIC can likely extract additional benefits [36]. Regardless, there are still various notable applications for FPGAs, e.g., SmartNIC designs for data centers [56].

necessary. Primarily, we should strip out unnecessary algorithmic complexity and enable as much parallelism as possible. Most commonly, this may manifest as either removing control flow or reducing runtime/memory complexity. One example of this pertains to how real-valued function primitives are implemented, as we mentioned at the end of Sect. 14.1. Future work should explore how forms of evaluation, selection, variation, local search, and multi-objective optimization may be simplified, and how any combination of such changes may affect solution quality.

3. *Explore the use of FPGAs for GP inference. (Easy to medium.)* As mentioned in Sect. 14.1, FPGA devices may be separately useful for implementing final GP models, where a hardware architecture employed during this stage could be completely different from any architecture used for training. Besides the example given for tree-based GP in Sect. 14.1, we highlight that employing FPGAs for *tangled program graphs (TPGs)* may be especially interesting, given that such a GP representation can already allow for models that are thousands of times more efficient than state-of-the-art neural networks [18, 38, 39].
4. *Make FPGAs more accessible to GP researchers. (Medium.)* As mentioned in Sect. 14.5.1, although many high-level tools for utilizing FPGA technologies now exist, effectively implementing hardware still often requires considerable digital design expertise. As such, most GP researchers would likely incur a significant learning curve if attempting to design specialized hardware. Future work should explore additional technologies, techniques, and tutorials for aiding the GP community in this regard. For an initial starting point, see [29, 68].
5. *Establish new performance/energy measures. (Easy.)* The traditional measure for GP system performance, *GP operations per second (GPops/s)*, is often insufficient in that it does not reflect time taken for more complex operations (e.g., local search) nor differences in solution quality. To alleviate such issues, future work should explore additional comparison measures, such as those given in [16]. Note that a single measure need not encompass all of performance, energy, and fitness, although it is possible [16].
6. *Reconsider Boolean and fixed-point domains. (Medium.)* Performance/energy enhancements with modern FPGAs for Boolean and fixed-point domains will likely be even more pronounced. Future work should reconsider these domains.

14.6 Conclusion

Rather than just play the “hardware lottery” [33], the GP community has the potential to forge its own destiny by designing specialized hardware through FPGA devices. The computational model of GP is exceptional in that it is often embarrassingly parallel, although additional algorithmic considerations are likely needed in order to fully exploit the possible benefits of specialized architectures. This chapter explored such concepts and laid the groundwork for future efforts involving FPGA devices. Overall, FPGAs may allow for faster and/or less costly GP runs, in which case it may

also be possible for better solutions to be found when allowing an FPGA to consume the same amount of runtime/energy as another computing platform.

References

1. Acun, B., Lee, B., Kazhamiaka, F., Maeng, K., Gupta, U., Chakkaravarthy, M., Brooks, D., Wu, C.J.: Carbon explorer: a holistic framework for designing carbon aware datacenters. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 2, pp. 118–132 (2023)
2. Alok, G.: Architecture apocalypse dream architecture for deep learning inference and compute - Versal AI core. Embedded World (2020)
3. Amazon: EC2 F1. <https://aws.amazon.com/ec2/instance-types/f1/> (2024)
4. AMD: Versal ACAP DSP Engine Architecture Manual (AM004). <https://docs.amd.com/r/en-US/am004-versal-dsp-engine/DSP58-Architecture> (2018)
5. AMD: Heterogeneous Accelerated Compute Cluster (HACC) Program. <https://www.amd.com/en/corporate/university-program/aup-hacc.html> (2024)
6. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring), pp. 483–485. Association for Computing Machinery, New York (1967). <https://doi.org/10.1145/1465482.1465560>
7. Banzhaf, W., Nordin, P., Keller, R., Francone, F.: Genetic Programming - An Introduction. Morgan Kaufmann, San Francisco (1998)
8. Bashir, N., Guo, T., Hajiesmaili, M., Irwin, D., Shenoy, P., Sitaraman, R., Souza, A., Wierman, A.: Enabling sustainable clouds: the case for virtualizing the energy system. In: Proceedings of the ACM Symposium on Cloud Computing, SoCC '21, pp. 350–358. Association for Computing Machinery, New York (2021). <https://doi.org/10.1145/3472883.3487009>
9. Brameier, M., Banzhaf, W., Banzhaf, W.: Linear Genetic Programming. Springer, New York (2007)
10. Burlacu, B.: GECCO'2022 symbolic regression competition: post-analysis of the Operon framework. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation, GECCO '23 Companion, pp. 2412–2419. Association for Computing Machinery, New York (2023). <https://doi.org/10.1145/3583133.3596390>
11. Burlacu, B., Kronberger, G., Kommenda, M.: Operon C++: an efficient genetic programming framework for symbolic regression. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, GECCO '20, pp. 1562–1570. Association for Computing Machinery, New York (2020). <https://doi.org/10.1145/3377929.3398099>
12. Cheang, S.M., Leung, K.S., Lee, K.H.: Genetic parallel programming: design and implementation. *Evol. Comput.* **14**(2), 129–156 (2006). <https://doi.org/10.1162/evco.2006.14.2.129>
13. Chitty, D.M.: Faster GPU-based genetic programming using a two-dimensional stack. *Soft. Comput.* **21**(14), 3859–3878 (2017). <https://doi.org/10.1007/s00500-016-2034-0>
14. Chromczak, J., Wheeler, M., Chiasson, C., How, D., Langhammer, M., Vanderhoek, T., Zgheib, G., Ganusov, I.: Architectural enhancements in Intel Agilex FPGAs. In: Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '20, pp. 140–149. Association for Computing Machinery, New York (2020). <https://doi.org/10.1145/3373087.3375308>
15. Coole, J., Stitt, G.: Adjustable-cost overlays for runtime compilation. In: 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, pp. 21–24 (2015). <https://doi.org/10.1109/FCCM.2015.49>
16. Cray, C., Burlacu, B., Banzhaf, W.: Enhancing the computational efficiency of genetic programming through alternative floating-point primitives. In: Parallel Problem Solving from Nature. Springer (2024). https://doi.org/10.1007/978-3-031-70055-2_20

17. Crary, C., Piard, W., Stitt, G., Bean, C., Hicks, B.: Using FPGA devices to accelerate tree-based genetic programming: a preliminary exploration with recent technologies. In: European Conference on Genetic Programming (Part of EvoStar), pp. 182–197. Springer (2023). https://doi.org/10.1007/978-3-031-29573-7_12
18. Desnos, K., Bourgoïn, T., Dardaillon, M., Sourbier, N., Gesny, O., Pelcat, M.: Ultra-fast machine learning inference through C code generation for tangled program graphs. In: 2022 IEEE Workshop on Signal Processing Systems (SiPS), pp. 1–6 (2022). <https://doi.org/10.1109/SiPS55645.2022.9919237>
19. Dobai, R., Sekanina, L.: Low-level flexible architecture with hybrid reconfiguration for evolvable hardware. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)* **8**(3), 1–24 (2015)
20. Dréan, G.: The chips industry: Moore and Rock’s laws. In: *The Digital Era 2: Political Economy Revisited*, pp. 125–135. Wiley Online Library (2019)
21. Eklund, S.: Time series forecasting using massively parallel genetic programming. In: *Proceedings International Parallel and Distributed Processing Symposium* (2003). <https://doi.org/10.1109/IPDPS.2003.1213272>
22. Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. In: *Proceedings of the 38th Annual International Symposium on Computer Architecture, ISCA ’11*, pp. 365–376. Association for Computing Machinery, New York (2011). <https://doi.org/10.1145/2000064.2000108>
23. Funie, A.I., Grigoras, P., Burovskiy, P., Luk, W., Salmon, M.: Run-time reconfigurable acceleration for genetic programming fitness evaluation in trading strategies. *J. Signal Process. Syst.* **90**(1), 39–52 (2018). <https://doi.org/10.1007/s11265-017-1244-8>
24. Funie, A.I., Salmon, M., Luk, W.: A hybrid genetic-programming swarm-optimisation approach for examining the nature and stability of high frequency trading strategies. In: *2014 13th International Conference on Machine Learning and Applications*, pp. 29–34 (2014). <https://doi.org/10.1109/ICMLA.2014.11>
25. García-Martín, E., Rodrigues, C.F., Riley, G., Grahn, H.: Estimation of energy consumption in machine learning. *J. Parallel Distrib. Comput.* **134**, 75–88 (2019)
26. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv. (CSUR)* **23**(1), 5–48 (1991)
27. Goribar-Jimenez, C., Maldonado, Y., Trujillo, L., Castelli, M., Gonçalves, I., Vanneschi, L.: Towards the development of a complete GP system on an FPGA using geometric semantic operators. In: *2017 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1932–1939 (2017). <https://doi.org/10.1109/CEC.2017.7969537>
28. Gustafson, J.L.: Reevaluating Amdahl’s law. *Commun. ACM* **31**(5), 532–533 (1988). <https://doi.org/10.1145/42411.42415>
29. Harris, S.L., Harris, D.: *Digital Design and Computer Architecture*. Morgan Kaufmann (2015)
30. Hennessy, J.L., Patterson, D.A.: *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th edn. Morgan Kaufmann Publishers Inc., San Francisco (2017)
31. Hettiarachchi, D.L.N., Davuluru, V.S.P., Balster, E.J.: Integer vs. floating-point processing on modern FPGA technology. In: *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 0606–0612 (2020). <https://doi.org/10.1109/CCWC47524.2020.9031118>
32. Heywood, M.I., Zincir-Heywood, A.N.: Register based genetic programming on FPGA computing platforms. In: Poli, R., Banzhaf, W., Langdon, W.B., Miller, J., Nordin, P., Fogarty, T.C. (eds.) *Genetic Programming*, pp. 44–59. Springer, Berlin (2000)
33. Hooker, S.: The hardware lottery. *Commun. ACM* **64**(12), 58–65 (2021). <https://doi.org/10.1145/3467017>
34. Intel: Intel Agilex™ M-Series FPGA and SoC FPGA Product Table (2015). <https://cdrdv2.intel.com/v1/dl/getContent/721636>
35. Intel: BFLOAT16 – Hardware Numerics Definition White Paper (2018). <https://www.intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-paper.pdf>

36. Jouppi, N., Young, C., Patil, N., Patterson, D.: Motivation for and evaluation of the first tensor processing unit. *IEEE Micro* **38**(3), 10–19 (2018). <https://doi.org/10.1109/MM.2018.032271057>
37. Kapre, N., Bayliss, S.: Survey of domain-specific languages for FPGA computing. In: 2016 26th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–12 (2016). <https://doi.org/10.1109/FPL.2016.7577380>
38. Kelly, S., Heywood, M.I.: Emergent solutions to high-dimensional multitask reinforcement learning. *Evol. Comput.* **26**(3), 347–380 (2018)
39. Kelly, S., Smith, R.J., Heywood, M.I.: Emergent Policy Discovery for Visual Reinforcement Learning Through Tangled Program Graphs: A Tutorial, pp. 37–57. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-04735-1_3
40. Kommenda, M., Burlacu, B., Kronberger, G., Affenzeller, M.: Parameter identification for symbolic regression using nonlinear least squares. *Genet. Program Evolvable Mach.* **21**(3), 471–501 (2020)
41. Koza, J.R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge (1992)
42. Koza, J.R., Bennett, F.H., Hutchings, J.L., Bade, S.L., Keane, M.A., Andre, D.: Evolving computer programs using rapidly reconfigurable field-programmable gate arrays and genetic programming. In: *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, FPGA '98*, pp. 209–219. Association for Computing Machinery, New York (1998). <https://doi.org/10.1145/275107.275141>
43. Krishnakumar, A., Ogras, U., Marculescu, R., Kishinevsky, M., Mudge, T.: Domain-specific architectures: research problems and promising approaches. *ACM Trans. Embed. Comput. Syst.* **22**(2) (2023). <https://doi.org/10.1145/3563946>
44. La Cava, W., et al.: Contemporary symbolic regression methods and their relative performance. In: Vanschoren, J., Yeung, S., (eds.) *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, vol. 1 (2021)
45. Lahti, S., Rintala, M., Hämäläinen, T.D.: Leveraging modern C++ in high-level synthesis. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **42**(4), 1123–1132 (2023). <https://doi.org/10.1109/TCAD.2022.3193646>
46. Lahti, S., Sjövall, P., Vanne, J., Hämäläinen, T.D.: Are we there yet? a study on the state of high-level synthesis. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **38**(5), 898–911 (2019). <https://doi.org/10.1109/TCAD.2018.2834439>
47. Leswing, Kif: Nvidia's latest AI chip will cost more than \$30,000, CEO says (2024). <https://www.cnn.com/2024/03/19/nvidias-blackwell-ai-chip-will-cost-more-than-30000-ceo-says.html>
48. L'Ecuier, P.: Maximally equidistributed combined tausworthe generators. *Math. Comput.* **65**(213), 203–213 (1996)
49. Maldonado, Y., Salas, R., Quevedo, J.A., Valdez, R., Trujillo, L.: GSGP-hardware: Instantaneous symbolic regression with an FPGA implementation of geometric semantic genetic programming. *Genet. Program. Evolvable Mach.* **25**(2), 18 (2024). <https://doi.org/10.1007/s10710-024-09491-5>
50. Martin, P.: A hardware implementation of a genetic programming system using FPGAs and Handel-C. *Genet. Program Evolvable Mach.* **2**(4), 317–343 (2001). <https://doi.org/10.1023/A:1012942304464>
51. McKay, R.I., Hoai, N.X., Whigham, P.A., Shan, Y., O'Neill, M.: Grammar-based genetic programming: a survey. *Genet. Program Evolvable Mach.* **11**(3), 365–396 (2010). <https://doi.org/10.1007/s10710-010-9109-y>
52. Miller, J.F.: Cartesian genetic programming: its status and future. *Genet. Program Evolvable Mach.* **21**(1), 129–168 (2020). <https://doi.org/10.1007/s10710-019-09360-6>
53. Moraglio, A., Krawiec, K., Johnson, C.G.: Geometric semantic genetic programming. In: Coello, C.A.C., Cutello, V., Deb, K., Forrest, S., Nicosia, G., Pavone, M. (eds.) *Parallel Problem Solving from Nature - PPSN XII*, pp. 21–31. Springer, Berlin (2012)

54. Nurvitadhi, E., et al.: Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17, pp. 5–14. Association for Computing Machinery, New York (2017). <https://doi.org/10.1145/3020078.3021740>
55. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming. Lulu Enterprises, UK Ltd (2008)
56. Putnam, A., et al.: A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro* **35**(3), 10–22 (2015). <https://doi.org/10.1109/MM.2015.42>
57. Quenon, A., Ramos Gomes Da Silva, V.: Towards higher-level synthesis and co-design with Python. In: Proceedings of the Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE'21). ACM New York (2021)
58. Robilliard, D., Marion-Poty, V., Fonlupt, C.: Genetic programming on graphics processing units. *Genet. Program Evolvable Mach.* **10**(4), 447–471 (2009). <https://doi.org/10.1007/s10710-009-9092-3>
59. Roser, M., Ritchie, H., Mathieu, E.: What is Moore's Law? Our World in Data (2023). <https://ourworldindata.org/moores-law>
60. Ruiz-Rosero, J., Ramirez-Gonzalez, G., Khanna, R.: Field programmable gate array applications—a scientometric review. *Computation* **7**(4) (2019). <https://doi.org/10.3390/computation7040063>
61. Salvador, R., Otero, A., Mora, J., de la Torre, E., Riesgo, T., Sekanina, L.: Self-reconfigurable evolvable hardware system for adaptive image processing. *IEEE Trans. Comput.* **62**(8), 1481–1493 (2013)
62. Sidhu, R.P.S., Mei, A., Prasanna, V.K.: Genetic programming using self-reconfigurable FPGAs. In: Lysaght, P., Irvine, J., Hartenstein, R., (eds.) *Field Programmable Logic and Applications*, pp. 301–312. Springer, Berlin (1999). https://doi.org/10.1007/978-3-540-48302-1_31
63. Skalicky, S., Monson, J., Schmidt, A., French, M.: Hot & Spicy: improving productivity with Python and HLS for FPGAs. In: 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 85–92 (2018). <https://doi.org/10.1109/FCCM.2018.00022>
64. So, H.K.H., Liu, C.: *FPGA Overlays*, pp. 285–305. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-26408-0_16
65. Spector, L., Robinson, A.: Genetic programming and autoconstructive evolution with the Push programming language. *Genet. Program Evolvable Mach.* **3**(1), 7–40 (2002). <https://doi.org/10.1023/A:1014538503543>
66. Stitt, G.: Are field-programmable gate arrays ready for the mainstream? *IEEE Micro* **31**(6), 58–63 (2011). <https://doi.org/10.1109/MM.2011.99>
67. Stitt, G., Gupta, A., Emas, M.N., Wilson, D., Baylis, A.: Scalable window generation for the Intel Broadwell+Arria 10 and high-bandwidth FPGA systems. In: Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '18, pp. 173–182. Association for Computing Machinery (2018). <https://doi.org/10.1145/3174243.3174262>
68. Stitt, G.: VHDL and SystemVerilog Tutorials. <https://stitt-hub.com/vhdl-and-systemverilog-tutorials/> (2024)
69. Strubell, E., Ganesh, A., McCallum, A.: Energy and policy considerations for modern deep learning research. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, pp. 13,693–13,696 (2020)
70. Tan, T., Nurvitadhi, E., Shih, D., Chiou, D.: Evaluating the highly-pipelined Intel Stratix 10 FPGA architecture using open-source benchmarks. In: 2018 International Conference on Field-Programmable Technology (FPT), pp. 206–213 (2018). <https://doi.org/10.1109/FPT.2018.00038>
71. Tessier, R., Pocek, K., DeHon, A.: Reconfigurable computing architectures. *Proc. IEEE* **103**(3), 332–354 (2015). <https://doi.org/10.1109/JPROC.2014.2386883>
72. Vašíček, Z., Sekanina, L.: Hardware accelerators for cartesian genetic programming. In: Genetic Programming: 11th European Conference, EuroGP 2008, Naples, Italy, March 26–28, 2008. Proceedings 11, pp. 230–241. Springer (2008)

73. Vašíček, Z., Sekanina, L.: Hardware accelerator of cartesian genetic programming with multiple fitness units. *Comput. Inf.* **29**(6+), 1359–1371 (2010)
74. Vipin, K., Fahmy, S.A.: FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Comput. Surv.* **51**(4) (2018). <https://doi.org/10.1145/3193827>
75. Yao, X.: Following the path of evolvable hardware. *Commun. ACM* **42**(4), 46–49 (1999). <https://doi.org/10.1145/299157.299169>
76. Yao, X., Higuchi, T.: Promises and challenges of evolvable hardware. *IEEE Trans. Syst., Man, Cybern., Part C (Appl. Rev.)* **29**(1), 87–97 (1999). <https://doi.org/10.1109/5326.740672>

Chapter 15

Interpretable Genetic Programming Models for Real-World Biomedical Images



Yuri Lavinas, Camilo De La Torre, Kévin Cortacero, Dennis G. Wilson,
and Sylvain Cussat-Blanc

Abstract The use of artificial intelligence in critical applications can be limited due to a lack of model understanding. Genetic programming (GP) offers an alternative to black-box methods like deep neural networks by evolving interpretable function representations. In this chapter, we present recent advances in the application of GP to the analysis of biomedical data, focusing on the challenges of evolving interpretable models for real-world datasets. We first detail Kartezio, a Cartesian Genetic Programming (CGP) approach that generates interpretable pipelines for biomedical image segmentation. We demonstrate that Kartezio can achieve competitive performance with state-of-the-art deep learning methods while requiring significantly smaller training datasets. This is particularly important in the biomedical domain, where fully annotated data is scarce. We further explore the use of small labeled datasets by proposing a data sampling mechanism based on active learning methods. Finally, real biomedical data often involves multiple data types, from numerical integers to complex multichannel images. We present the Multimodal Active Genetic Evolution (MAGE) algorithm, which extends CGP's representation to include functions of different types, opening new possibilities beyond segmentation. Through this work, we demonstrate that GP can be a powerful tool for high-dimensional data analysis, providing interpretable models that can be used in critical applications.

Y. Lavinas (✉) · C. De La Torre
IRIT - CNRS UMR5505, University Toulouse Capitole, Toulouse, France
e-mail: yclavinas@gmail.com

C. De La Torre
e-mail: camilo.de-la-torre@ut-capitole.fr

K. Cortacero
CRCT - INSERM, Toulouse, France
e-mail: kevin.cortacero@inserm.fr

D. G. Wilson
ISAE-SUPAERO, Toulouse, France
e-mail: dennis.wilson@isae.fr

S. Cussat-Blanc
IRIT - CNRS UMR5505, University Toulouse Capitole, Institut Universitaire de France, Toulouse, France
e-mail: sylvain.cussat-blanc@irit.fr

15.1 Introduction

The field of biomedical image analysis has experienced a revolution with the usage of Neural Networks, particularly Deep Learning (DL) techniques, displaying a similar trend in the broader domain of Computer Vision. DL algorithms exhibit remarkable performance across various image classification and segmentation tasks within medical disciplines such as dermatology, radiology, and pathology [11, 12], occasionally surpassing human experts. Nonetheless, DL methodologies encounter two primary limitations. Firstly, they are regarded as black-box systems, as the decision-making processes of deep Neural Networks often lack interpretability, which poses a challenge in critical domains like medicine since understanding of model choices is imperative. Secondly, DL approaches need substantial amounts of annotated data, a resource-intensive and costly task typically carried out by busy experts, that reduces the availability of larger datasets for training purposes.

To deal with this interpretability issue, we focus on evolving programs for instance segmentation with Cartesian Genetic Programming (CGP) [36]. CGP is a Genetic Programming variant for evolving direct and acyclic graphs, that are indexed by Cartesian coordinates. Evolution defines how to connect the nodes of the graphs and the function of each node. The idea of using CGP for image processing is to rely on human-developed methods in image analysis. CGP develops solutions based on a set of mathematical functions that, when executed, process certain inputs to produce an expected output. Herein, we summarize recent findings of our CGP approach on image analysis tasks, named *Kartezio*, that creates explainable and interpretable image processing pipelines.

First, we show that *Kartezio* is an effective approach to address some of the limitations in specific interpretability, of Deep Learning (DL) [9, 27–29], with results competitive to DL. Moreover, *Kartezio* can be trained in just a few hours, using less data: *Kartezio* generates effective models using a dataset consisting of only 10 images while the DL methods require 20 times more images to train effective models [9].

Then, we study how to improve the generation of programs using small labeled datasets. Generating effective programs using a small amount of labeled data during evolution is important, since the amount of biological data available is extremely limited, mainly due to complex features and expensive to collect and annotate. For that, we use a popular approach used in machine learning (ML) called Active Learning (AL) [13], for identifying useful sample images via a sampling method.

We sampled the subset of images that will be part of the training dataset used in evolution via the following sampling mechanism that maximizes an uncertainty metric. The idea of using uncertainty as the metric to select data is that it can direct evolution to use images that are the most enlightening to the current models in training [21, 28], since they have information not yet well processed by the models [21, 28]. We show that it is possible to improve the convergence speed of *Kartezio* using AL while maintaining the size of the models, indicating that the models are effective and interpretable.

The analysis of biomedical data relies on integrating and interpreting diverse data types, from numerical integers to textual strings, and extending to complex multichannel images which not only increase the search space but also expand the number of local optima, complexifying the structure of the fitness landscape. Given this multimodality, we briefly study how to extend Kartezio to integrate multiple function libraries, each specialized for a distinct return type, such as integer, float, and arrays. With this extension, having multimodal libraries enables the combination of data-dependent functions, leading to more sophisticated pipelines. For example, calculating the average of all pixel values in an image is not a simple task based solely on scalar pixel input values and would be exceedingly challenging for a single-type CGP [17].

15.2 Background

Most work in the area of Computer Vision uses black-box approaches, such as artificial Deep Neural Networks. However, humans find it hard to describe the reason behind some of the choices made by Neural Network models and then to explain these choices to users and experts, thus lacking interpretability [41].

One way to complement these black-box methods is to use methods that are explainable by design. The class of Genetic Programming (GP) approaches is one among such methods, as GP optimizes interpretable forms of function representations such as trees or graphs. Graph-based Genetic Programming methods, like Cartesian Genetic Programming, naturally tend toward small function graphs, increasing their interpretability [37, 51].

CGP has been successfully applied in multiple domains. For example, CGP has been extended to build an integral image processing algorithm using high-level operations [5]. Also, CGP has also been applied in Computer Vision tasks such as for controlling agents to play ATARI games [51], and in image processing tasks, such as biomedical image segmentation and object detection in robotics [9, 18]. In addition, there have been works on hybrid approaches that combined Neural Networks and CGP [1, 47].

15.2.1 *Cartesian Genetic Programming*

CGP [36] builds upon the concepts of GP [26] by adopting a directed acyclic graph (DAG)-based representation of computer programs, as shown in Fig. 15.1, (Figure from [2]). In this representation, nodes combine inputs and a given operation, and these nodes are derived from a vector known as the genotype. Initially, the nodes were organized into a two-dimensional grid (rows and columns)—hence the term “Cartesian”. Each node could only be connected to another node in a preceding

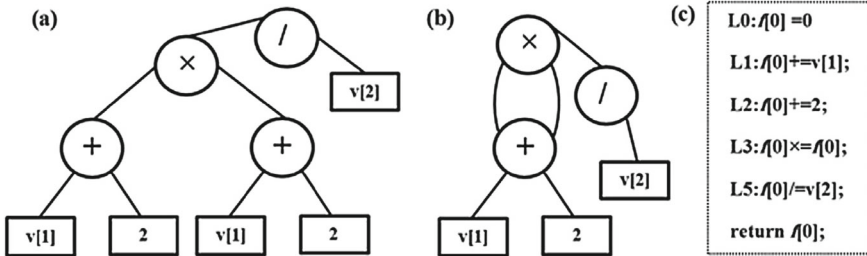


Fig. 15.1 A comparison of a program structure evolved by **a** tree-based, **b** graph-based GP, and **c** linear GP for the expression of $y = f(v) = \frac{(v_1+2)^2}{v_2}$

column. However, this grid concept was later abandoned in favor of using a single row, thereby defining the genotype's "dimension" by the number of nodes alone.

The directed graph is obtained by sequentially connecting a given node's outputs to its following node's inputs, and finally to the output nodes, making the resulting graph not necessarily a tree. Another important characteristic is the graph's acyclic nature, due to each node being always connected to a preceding one, thereby eliminating the possibility of intrinsic loops. Within the graph, certain nodes may not belong to the resultant structure and can therefore be disregarded. The graph that remains can be termed the "active" graph, comprising only the "active" nodes. Typically, CGP employs the $1 + \lambda$ evolutionary algorithm (EA) and usually does not utilize the Crossover operation, although a few methods have been proposed [8, 24].

The genotype in CGP is a numerical vector, typically of fixed size. This vector encodes both functional and output nodes. The nodes within the genotype are represented as sub-vectors of equal size, which are broken down into: (a) the ID of the function used, (b) the nodes that serve as inputs for the given node, and (c) any additional parameters required for the function. The genotype is completed by the output nodes, which point to either functional nodes or direct inputs. While the genotype can be easily modified by changing any of its values, the implications of such modifications only become clear when the genotype is decoded into a phenotype.

In CGP, the phenotype is a DAG that is assembled, connected, and parameterized based on the nodes specified by the genotype. The decoding operation in CGP starts with each output node pointing to a functional node. Each functional node, in turn, points to one or more preceding nodes depending on the arity of the function in question. This operation is recursively repeated until input nodes are reached, which eventually results in the final active graph. The combination of an adapted decoding method and a novel genotype encoding can give rise to new CGP models.

In standard CGP, a Library is a collection of functions, also referred to as Primitives. Importantly, primitives must always be deterministic—given the same inputs and parameters, they must always produce the same output. In models developed for image processing, where the library of functions comprises filters or operations on images (typically taken from the OpenCV Library [6]). This has resulted in variants

of CGP such as CGPIP [5, 19, 32] and ALIFCGP [39], which restricts its operations to binary and morphological operators.

Mutation plays a fundamental role in CGP, as it serves as the primary mechanism for altering genotypes during evolution. In CGP, point mutation of a gene involves changing its value. This could involve modifying connections, the function, the function parameters, or the direct output connections. Mutation rules are implicitly directed and constrained by the genotype as well as the type of model.

In addition to the specific effects of a mutation, various strategies can be employed. Miller summarizes these strategies [35], building on the work of Goldman and Punch [15]. Their aim is to minimize the evaluation of genotypes that have phenotypes identical to those of previous generations. In other words, if the effects of mutations do not influence the phenotype (active graphs and parameters), the predicted outputs and hence the fitness obtained will remain the same.

15.2.2 *Cartesian Genetic Programming for Image Processing*

CGP is a flexible evolutionary paradigm, that has found applications across various domains, such as image processing. In this, CGP presents a promising avenue for automatically synthesizing image filters or transformations.

Sekanina and Martinek [34, 43, 45] proposed implementations on FPGA hardware. Harding [16] drastically improved the computation time performance by optimizing the final image filter computation on GPU. A major part of the use of CGP for image processing is synthesized in 2011 [44].

CGPIP [5] was introduced in 2012 to extend the utility of CGP to build an integral image processing algorithm using high-level operations. For this, the main function library comprises pre-existing filters, applying directly to an entire image and returning a filtered image. The diverse applications include the segmentation and classification of rocks on Mars [32], classification of cell mitosis [19], and detection of simple objects in robotics [19]. Another application of CGPIP in robotics allows robots to detect their own hands [31] and improve object detection [30]. Other applications of CGP in computer vision tasks include controlling agents to play ATARI games [51], biomedical image segmentation, and object detection in robotics [9, 18, 27].

15.2.3 *Kartezio: CGP for Image Processing*

Figure 15.2 displays the Kartezio pipeline, which uses CGP for image segmentation. This genotype is a sequence of integers known as genes (each one represented as a box containing a single integer) that are organized into nodes. A node is composed of a single function drawn from the function library, at least one connection, and optional parameters. Moreover, in this implementation, the end nodes of CGP are connected to a fixed endpoint, selected based on human expertise. This endpoint is

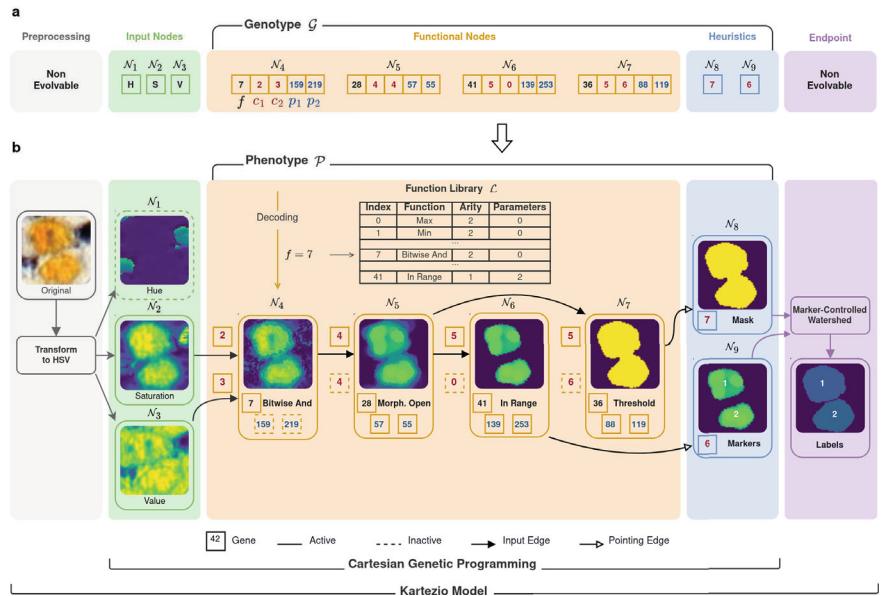


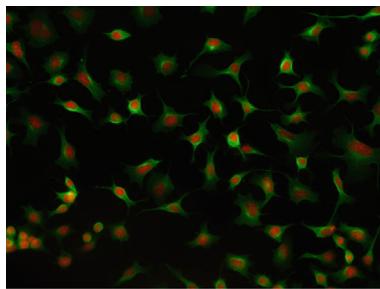
Fig. 15.2 The Kartezio pipeline. The main component is CGP using a library of image processing functions. The CGP genotype is a sequence of integers that are organized into nodes. Each node is composed of a function, connections, and optional parameters. The output nodes are connected to a fixed endpoint, the Watershed Transform, which generates the segmentation mask. Figure from [9]

useful since it allows CGP to obtain insight given by a computer vision expert. We use the Watershed Transform [4] as the endpoint. Thus, our CGP has 2 outputs, shown in the light blue are on the right side of Fig. 15.2, corresponding to the mask and markers necessary for the Watershed Transform endpoint, shown in the purple area, on the rightmost side of the figure. Finally, we use image processing functions mostly from OpenCV and Skimage (Scikit-Image) [50], which apply programs directly to images. Only a few nodes, the “active” nodes, are used, as they are actually connected to the output of the program graph, directly or indirectly. Other nodes with no connections to the output are called “inactive” nodes. The outputs of a program can come from any node and are determined by the evolutionary process.

15.3 Evaluation Settings

Average Precision (AP) is defined by the average precision $AP = TP / (TP + FP + FN)$, where TP means true positives, FP means false positives, and FN means false negatives [46]. We use AP as our fitness function with a threshold of 0.5 to determine the true positives of the predicted mask. As is commonly done in multi-class image

Fig. 15.3 A target image from the CELLPOSE dataset



segmentation, we use the Mean Average Precision (MAP), which is the average of all APs of each instance. *Intersection over Union (IoU)* evaluates the performance of an algorithm in detecting objects within an image by comparing the ratio among areas occupied by different objects [40] by measuring the quality of a predicted mask. A mask covering exactly the annotation will produce an Intersection equals to Union. Higher values suggest a higher overlap among such objects (the maximum is 1.0).

We use the CELLPOSE dataset [46] which consists of 100 images of fluorescent-labeled protein of cultured neuroblastoma cells stained with phalloidin FITC and DAPI. For a fair comparison, we follow the work in [46], where the data is split into 89 images for training and 11 for testing. Figure 15.3 shows a target image from this dataset used for testing.

15.4 Kartezio Versus Deep Learning

To evaluate the performance of this approach against state-of-the-art DL models, we compared Kartezio to Cellpose [46], a recently released DL approach for image segmentation in biology with both specialist and generalist models, as well as to two previously released approaches, Mask R-CNN [22] and StarDist [42]. As described below, we first evaluated Kartezio on the CELLPOSE dataset and compared it to the reported performance of the Cellpose specialist model (Fig. 15.4) [46].

We used randomly selected subsets of these 89 images (ranging from 1 to 89 images) to train Kartezio from scratch, with the parameters shown in Table 15.1, and subsequently evaluated its performance on the previously unseen CELLPOSE test data. Figure 15.4 shows the scores obtained on the 11-image test dataset after first training Kartezio on datasets of different sizes. Dotted lines indicate the results obtained by the other three methods as reported in Stringer et al. [46], when trained on all 89 images. It is interesting to note that Kartezio frequently outperformed [22] and [42] even when trained on much smaller datasets (e.g. on average, Kartezio matched Mask R-CNN when trained on as few as 6 images, and matched StarDist when trained on only 3 images). This illustrates the capacity of Kartezio to extrapolate from a very small set of training images and yet achieve comparable or superior

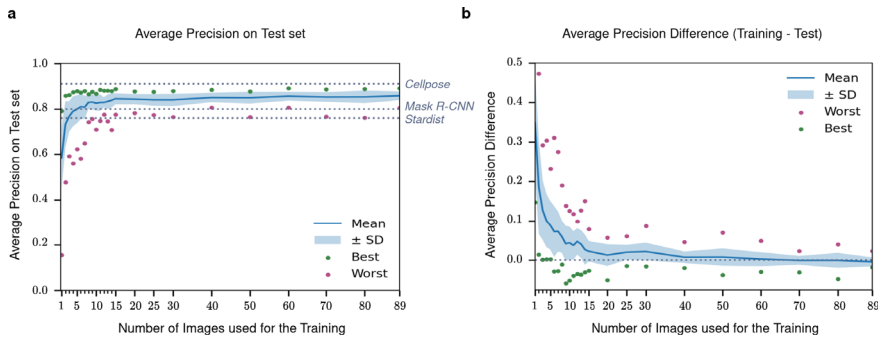


Fig. 15.4 Kartezio was trained on 1 to 89 images from the CELLPOSE dataset. **a** AP scores are shown for the indicated number of training images (mean $\pm$ SD of $n = 35$ models per condition; each model represents an independent experiment). **b** Mean differences in score between training and test datasets. The dotted line indicates an optimal difference of 0. Dots indicate best (green) and worst (red) scores obtained by Kartezio for the indicated number of training images

Table 15.1 Parameters used

Parameter	Value
Number of nodes	30 nodes
Offspring size, λ	5
Inputs	2
Outputs	2, mask and markers
Mutation of function nodes	0.15
Mutation of outputs nodes	0.20
Training generations	20,000
Independent executions	35

accuracy compared to state-of-the-art DL approaches. Our results also show that the best score obtained by Kartezio was 0.89, a value comparable to the reported best performance of Cellpose (0.91). Increasing the size of the training set in 10-image increments from 30 to 80 images did not perceptibly change Kartezio’s performance.

15.5 Dynamic Image Sampling in Kartezio

Dynamic data sampling is one of the most important components of Active Learning (AL) methods. AL is frequently used in Deep Learning in the context of processing biomedical data [13, 38] with most of the work on AL focusing on finding the metric that leads models to the highest performance [13]. Interestingly, some papers suggest random sampling as a strong baseline [7, 25].

Algorithm 1 AL in CGP (ABS-Kartezio)

```

1: Initialize  $(1 + \lambda)$  CGP.
2: while Stopping criterion is not met do
3:   Evolve CGP with the training dataset.
4:   Update elite solution.
5:   Calculate uncertainty metric on the images not in the training set.
6:   Add one image to the training dataset, the image with the highest uncertainty metric value.
7: end while

```

The efficacy of Active Learning in GP for symbolic regression tasks has been widely demonstrated, in works that focus on reducing the number of such evaluations [14] and on creating smaller, balanced datasets by recursively keeping the most “meaningful” exemplars [49] to studies on improving the rate and consistency at which well-performing solutions are found while reducing the required number of training samples [20, 21]. For classification problems, Hamida et al. [3, 23] showed how different sampling methods studied across the years affect the performance of GP. Yet, we did not find works that combine AL and GP in the biomedical domain. Recently, we have shown that AL [28, 29] methods can provide performance improvements in biomedical image processing.

By defining CGP with Adaptive Biomedical-image Sampling (ABS-Kartezio) template, we propose a method for instantiating and designing sampling techniques variants, as shown in Algorithm 1. The main difference to standard CGP is that instead of using a fixed training dataset during evolution, our template uses a smaller dataset that is selected during evolution, given the different metrics (explained below). We selected the subset of images to be part of the training dataset used in evolution via sampling mechanisms. Two of these mechanisms sample data based on uncertainty-related information, while the other mechanism samples images based on values taken from a uniform distribution.

Uncertainty counts the pairwise differences between the pixels of the predicted masks for each program on an image. If two pixels have a different label between two masks, m_1 and m_2 , a value of 1 is assigned to that pixel; otherwise, a value of 0 is applied. For an image of pixel dimensionality (i, j) , the overall uncertainty for a given image is the average uncertainty of all the pairwise uncertainties between each program. The metric ranges from 0 to 1.

$$\text{uncertainty} = \sum_{i,j}^{\text{Pixels}} k = \begin{cases} 1, & \text{if } m_{1(i,j)} \neq m_{2(i,j)} \\ 0, & \text{otherwise.} \end{cases}$$

The Weighted Uncertainty doubles the uncertainty measurement if one pixel is labeled as zero and the other pixel is labeled with a non-zero value. This assumes higher uncertainty should be assigned if models disagree on whether a pixel is foreground (non-zero) or background (zero). Where models agree on a pixel being foreground but disagree on which label it should be assigned, an uncertainty of 1 is given. The weighted metric ranges from 0 to 2.

Table 15.2 Parameters used

Parameter	Value
Number of nodes	30 nodes
Offspring size, λ	5
Inputs	2, α -tubulin and DAPI channels
Outputs	2, mask and markers
Mutation of function nodes	0.15
Mutation of outputs nodes	0.20
Images processed	1,000,000
Training generations	100
Independent executions	30

$$\text{Weighted} = \sum_{i,j}^{\text{Pixels}} k = \begin{cases} 2, & \text{if } m_{1(i,j)} \neq m_{2(i,j)} \text{ and } m_{1(i,j)} = 0 \text{ or } m_{2(i,j)} = 0, \\ 1, & \text{if } m_{1(i,j)} \neq m_{2(i,j)} \text{ and } m_{1(i,j)} \neq 0 \text{ and } m_{2(i,j)} \neq 0, \\ 0, & \text{otherwise.} \end{cases}$$

Random Uniform sampling serves as a baseline. Sampling is done using uniformly distributed values to select one image at each step. There is not any information gathered from the parallel runs of CGP.

To calculate uncertainty in CGP, we use a group of parallel Kartezio runs. The idea of maximizing uncertainty relies on our intuition that parallel runs of CGPs can increase the diversity of elite programs. Model diversity can be exploited to guide the collection of informative data [21]. The parallel runs are employed to estimate uncertainties only. Thus, for generating segmentation masks and all metrics related to the performance of ABS-CGP we choose from the parallel runs one program, the one with the highest fitness value on the training data.

Since we study the effect of sampling images from the dataset during evolution, we have a different number of images processed by ABS-Kartezio and standard Kartezio, at each generation. Thus, we use the number of images processed during evolution as the termination criterion.

ABS-Kartezio builds the dataset for training using the dynamic data sampling methods and standard CGP uses all 89 images available in the training set. We run all CGP variants with the parameters shown in Table 15.2, the same as in [9]. ABS-Kartezio adds new data to the dataset for training every 100 generations. For statistical purposes, 30 independent runs are done for each variant. We test three configurations on the number of parallel runs. For Kartezio and ABS-Kartezio with uniform metric, the number of parallel runs are 1, 5, 10, and 15. For the uncertainty-based metrics, the number of parallel runs are 2, 5, 10, and 15. We run ABS-Kartezio with the parameters from Table 15.2 while Kartezio uses the same parameters as before, with the exception of the stop criterion, that here is the number of images processed from Table 15.2.

Table 15.3 AP performance of parallel runs of Kartezio

	Parallel runs	Mean (standard deviation)
Kartezio	1	0.817 (0.036)
Kartezio	5	0.801 (0.032)
Kartezio	10	0.786 (0.035)
Kartezio	15	0.778 (0.026)

Table 15.4 Mean AP performance of parallel runs of ABS-Kartezio with different sampling mechanisms given 30 independent executions for each configuration

Sampling mechanism	Parallel runs	Mean (standard deviation)
Uncertainty	2	0.836 (0.018)
Uncertainty	5	0.839 (0.019)
Uncertainty	10	0.840 (0.018)
Uncertainty	15	0.837 (0.016)
Uniform	1	0.839 (0.025)
Uniform	5	0.845 (0.016)
Uniform	10	0.845 (0.017)
Uniform	15	0.848 (0.014)
Weighted uncertainty	2	0.840 (0.028)
Weighted uncertainty	5	0.840 (0.019)
Weighted uncertainty	10	0.839 (0.007)
Weighted uncertainty	15	0.840 (0.015)

We systematically examine and interpret the outcomes of image segmentation into cells versus background obtained through experimentation based on overall performance and convergence speed, images sampled, and number of nodes used over evolution. We highlight that all metrics and measures are calculated based on the highest-performing program of the parallel runs of the CGPs variants.

We analyze the impact of using parallel runs of Kartezio on the test data performance, with summary results shown in Table 15.3. Running Kartezio with more different parallel runs than the minimum (one) led to worse performance. Despite our initial expectation, that more runs in parallel would benefit Kartezio due to more diversity, the results show a counter-intuitive trend. Using parallel runs leads to less focus on exploration of potentially good search paths.

Table 15.4 shows the impact on performance on the test data of using parallel runs in ABS-Kartezio. The main result is that data sampling positively impacts the performance of CGP, since all methods found better performance than traditional CGP that uses all data available for training. That is not a surprise, since sampling data during evolution allows the programs to explore more carefully individual features of images, without overfitting. Also, combining dynamic data sampling with parallel ABS-Kartezio runs increases the performance of Kartezio.

15.5.1 *Kartezio Versus ABS-Kartezio*

The highest-performing ABS-Kartezio configurations are highlighted in **bold** in Table 15.4. We can verify the convergence speed of such algorithms in the sequential box plots shown in Fig. 15.5. ABS-Kartezio clearly converges faster than traditional CGP, while all ABS-Kartezio showed a relatively high performance from the very beginning. The performance of ABS-Kartezio with any of the metrics shows a statistical difference to Kartezio, with $p\text{-value} < 0.05$, based on the Student t-test.

Figure 15.6 shows the number of nodes of Kartezio and the highest-performing ABS-Kartezio for each sampling method. All CGP variants work with small programs during the whole evolution process. The size of the programs evolved by

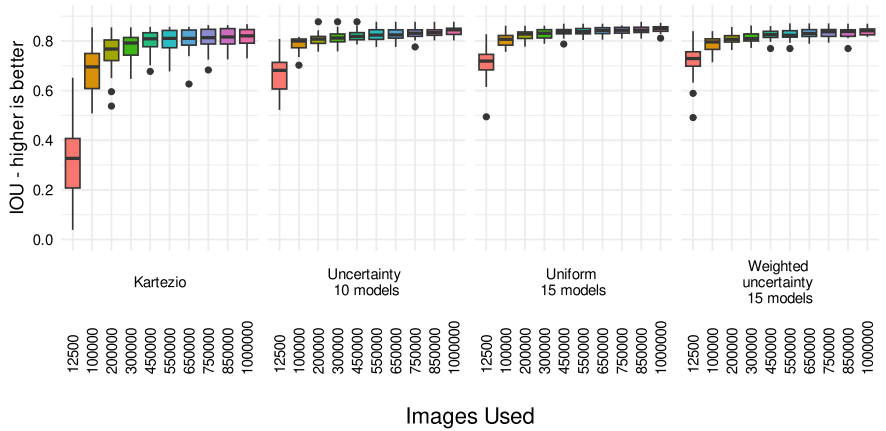


Fig. 15.5 Dynamically sampling images for training helps Kartezio to achieve high performance faster. We show the best-performing configuration of CGP and ABS-Kartezio with: uncertainty, uniform and weighted uncertainty

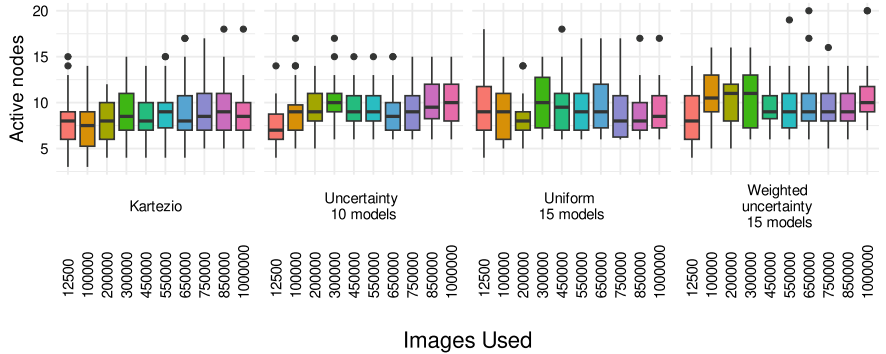


Fig. 15.6 Size of the programs generated given the number of active nodes

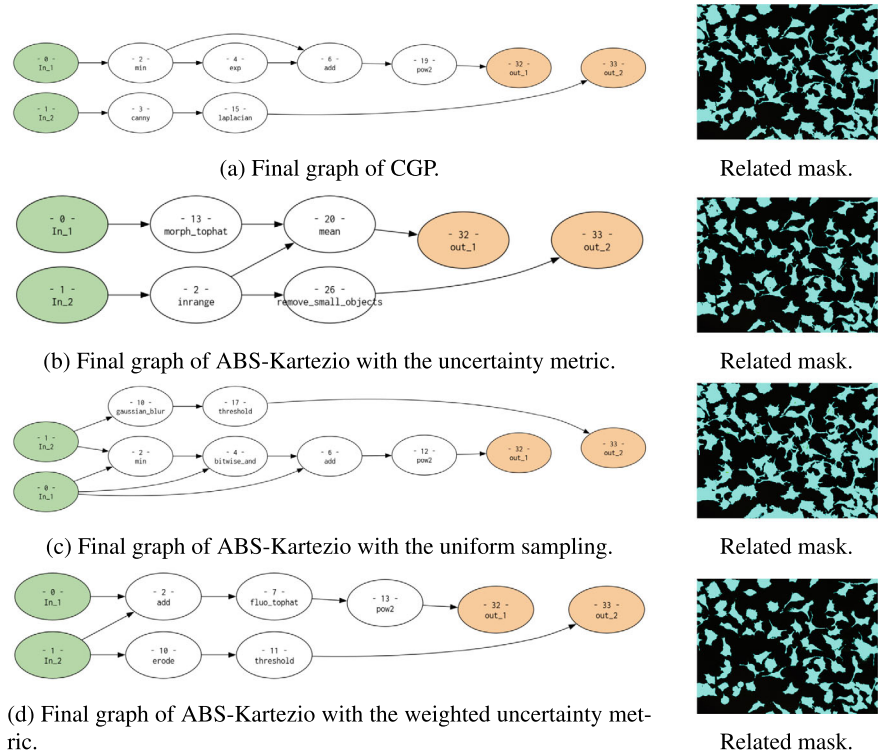


Fig. 15.7 Example of interpretable graphs by ABS-Kartezio used to create the segmentation mask for the image in Fig. 15.3

ABS-Kartezio varies given the sampling method, although all are working with programs slightly bigger than Kartezio. Figure 15.7 shows the final models generated by CGP, Fig. 15.7a, and by ABS-Kartezio, Fig. 15.7b, c, and d; and their related outcome segmentation obtained using those models. The segmentation models produced by ABS-Kartezio show different levels of complexity and interpretability. These figures give strong support that the models generated are overall simple and likely very interpretable.

Figure 15.8 shows the most frequently selected images: 0, 20, and 21; and, for comparison, the least frequently selected ones: 78, 45, and 63. Figure 15.8a, b, and c show that the size, shape, and brightness of the cells vary considerably, possibly indicating different details of the cells present in these images. This variation in cells indicates more complex data, which is a reasonable explanation for why these images were frequently sampled. Our analysis goes in agreement with observations from the biological field, where images with more cells are harder to analyze because the density of the cell often leads to cell membranes getting in contact, and possibly covering between cells (cells on top of each other). Additionally, images with high cell densities have a higher likelihood of showing rare events, which can make analysis

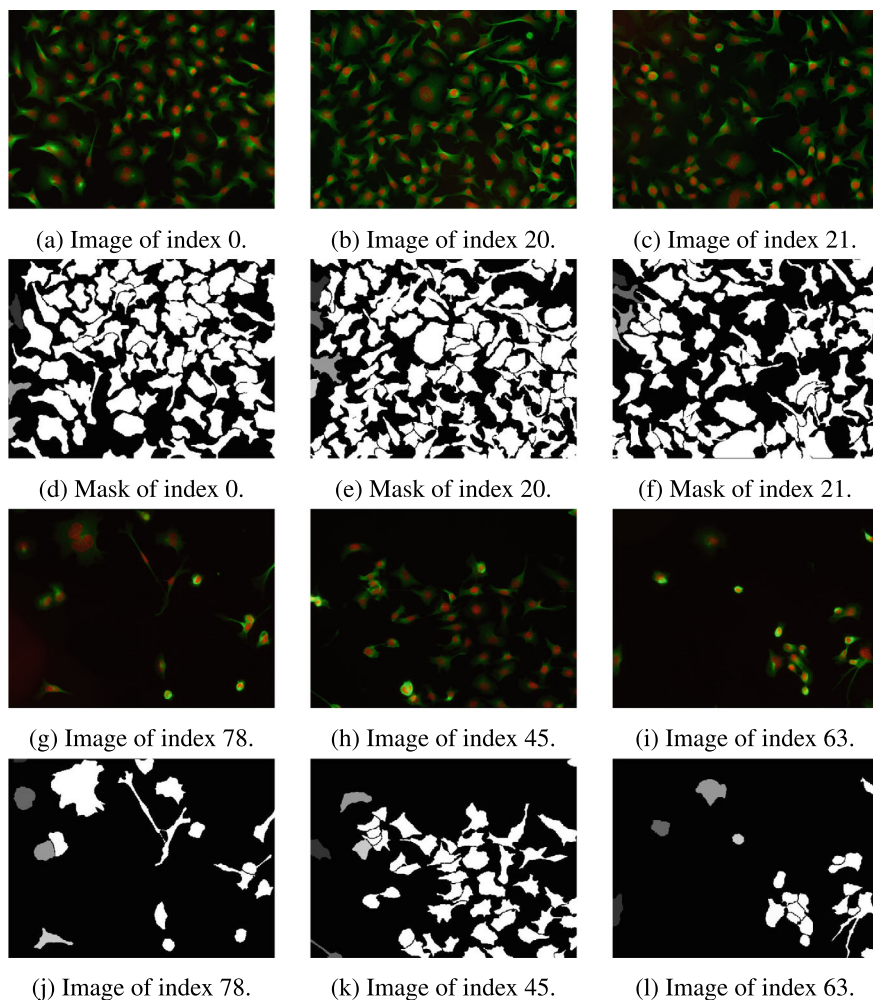


Fig. 15.8 Frequently sampled images (top, followed by the associated mask) and least frequently sampled images (bottom, followed by the associated mask). The most frequently sampled images lead to more uncertainty for ABS-Kartezio, since there are many cells to be identified that vary in size, shape, and color

more difficult. One of such events are mitosis, where a cell produces two identical nuclei in preparation for cell division, increasing cell density

Looking at the bottom images, Fig. 15.8g, h, i, of 15.8, however, it is possible to come up with a different scenario. The clearest distinction is that the number of cells is drastically smaller in this case, which, by itself, leads to a decrease in uncertainty values of the programs within the parallel ABS-Kartezio runs. This is because our metrics indirectly take into consideration the number of cells present in an image thus

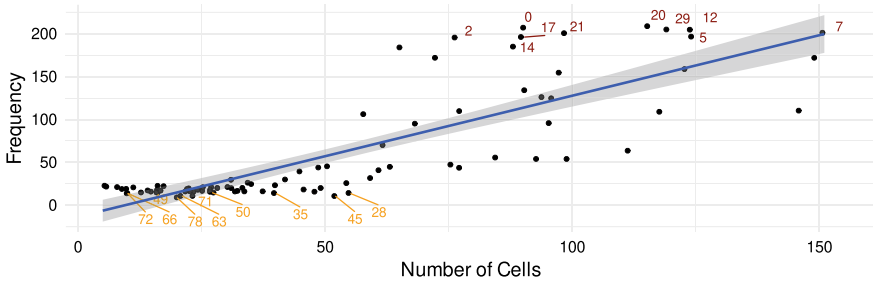


Fig. 15.9 Image sampled frequency versus number of cells per image. The regression line in black shows some correlation between the variables. Numbers in dark red show the index of the ten most frequently sampled images and the numbers in orange show the index of the ten least frequently sampled images

limiting the total uncertainty estimation of the program of the parallel ABS-Kartezio runs. Overall, we see several differences between the two groups of images, revealing possible insights into the characteristics of this dataset

When we look at the scatter-plot on the image sampled frequency versus the number of cells in images, Figure 15.9, we see that more images are selected as the number of cells increases. These results are based on the aggregate outputs of ABS-CGP with uncertainty-based metrics over 60 independent executions (30 runs for each metric of these metrics, the results of traditional CGP and ABS-CGP with Uniform sampling are not shown since they do not sample images), since these metrics are the ones that capture information about images. Some of the most frequently selected images 0, 20, and 21 have 175, 175, and 174 cells, respectively. On the other hand, some of the least frequently selected, images 78, 45, and 63, have 20, 52, and 21 cells, respectively, a much lower number of cells. That supports our argument that the number of cells in an image leads to more uncertainty in the programs in the parallel ABS-Kartezio runs; however, it is clear that this alone is not the only cause of uncertainty.

Given the fast convergence of the solutions shown by ABS-Kartezio and the preference for sampling some specific images, we test if using information about the frequency of sampled images could benefit the performance of CGP. We argue that these sampled images are highly informative and the ones that pose more challenges to the models, thus requiring more training effort. We compare: (1) ABS-Kartezio with the uniform metric and 15 models, (2) the 10 most frequently sampled images ABS-Kartezio selected before training, (3) the 10 least frequently sampled images, and (4) Kartezio that uses 10 images randomly selected before training.

Figure 15.10 shows the median AP convergence results. The performance of ABS-Kartezio (mean: 0.85 and standard deviation: 0.01) shows that using dynamic sampling leads to high-performing models that can generalize well to the test data (as shown by the low standard deviation), suggesting we need to find a fine balance in the choice of representative images to be used in training. Interestingly, having a dataset composed of 10 randomly selected images can achieve high final results

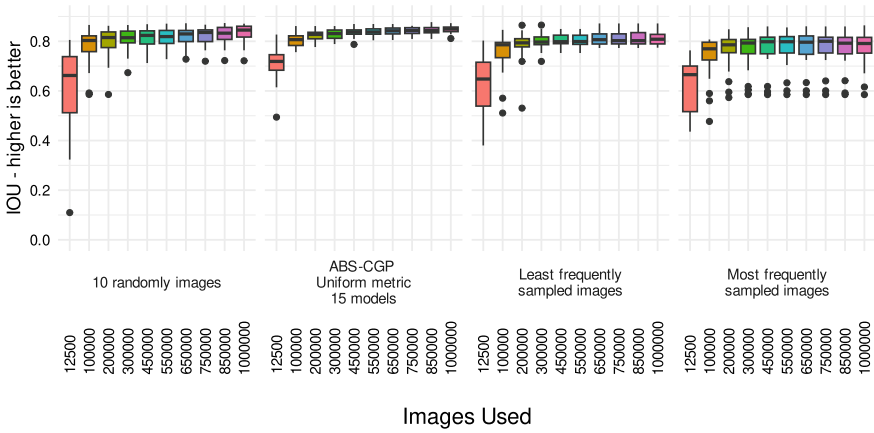


Fig. 15.10 Median convergence performance. ABS-Kartezio converges fast, to high values with lower standard deviation

(mean: 0.83 and standard deviation: 0.03), however, with a higher standard deviation, meaning randomly selecting images before the search progress starts is sensitive to the data sampling. The CGP variants that use the most and least frequently sampled images are the worse-performing algorithms that converge early during the search process, before half of the search, showing that they are not able to generalize well to the test data. These findings exhibit the significance of dynamic sampling and AL concepts in optimizing performance for this problem domain.

15.6 Multimodal Kartezio

Kartezio employs a single function library and a single data type. In this configuration, many functions require parameter values. Since these parameters are scalars, they cannot be directly used as inputs in CGP and must instead be encoded in the genotype. To overcome this limitation, MT-CGP was proposed [17]. This CGP variant still uses a single function library but in an augmented fashion. The function library incorporates multimodal functions, that is, functions of mixed types. MT-CGP overcomes the monotype limitation in CGP; however, now some graph connections are prohibited. One would hope that evolution connects and arranges the sequence of functions on its own, gradually resolving function mismatching. However, we found that that is not always the case, as much genetic material and computational effort is lost in evaluating incoherent phenotypes. Therefore, we introduced MAGE (Multi-Adaptive Graph Evolution) as a way of guiding evolution based on the function's signatures [10]. MAGE incorporates a multi-chromosome representation, in this way, multiple function libraries, specialized on a single return type, are separated. This separation enables a type-verification step without the need for evaluation. In other

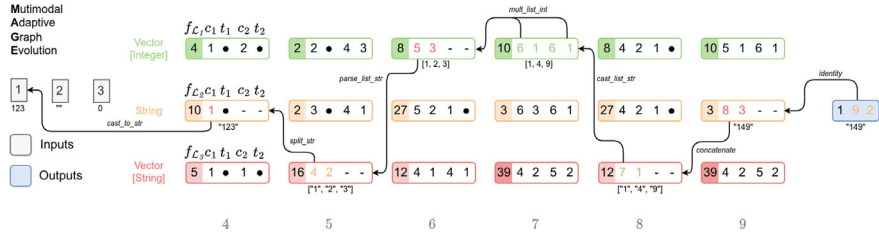


Fig. 15.11 The MAGE representation. Functions are organized by type and connections are protected to ensure type safety. In this example, the first, second, and third rows of functions represent Vector[Integer], String, and Vector[String] types, respectively. Figure from [48]

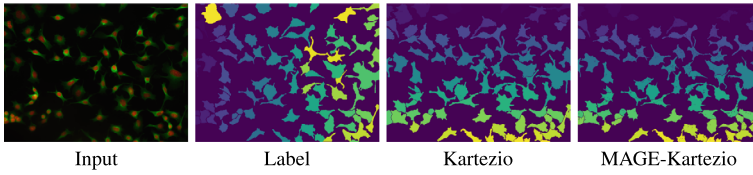


Fig. 15.12 An example of instance segmentation, with the labels provided by Kartezio and MAGE-Kartezio

words, when connecting to a node, we can know, beforehand, of which type will its output be. As such, we can allow only the mutations that yield coherent phenotypes, thus, aiding evolution in the search process.

Hence, we extended Kartezio to work with a scalar chromosome to dynamically compute function parameters as part of the evolved program and named this variant MAGE-Kartezio. Figure 15.11 presents a visual example demonstrating MAGE-Kartezio's explicit encoding approach. The key components of MAGE are depicted, including the variety of function libraries, each linked to a distinct chromosome, and the interconnections spanning across types.

We compare MAGE-Kartezio directly to Kartezio, where CGP employs constant, evolved parameters and MAGE-Kartezio potentially calculates dynamic scalars as parameters. The hyperparameters are identical: a mutation rate set to $\mu = 0.05$ for the function genes, $\nu = 0.1$ for the output genes, $\eta = 30$ nodes, and $\lambda = 5$ children. For this study, the fitness function employed is a composite of Average Precision (with a threshold of 0.5) (AP50) and Intersection over Union (IoU) to smooth the fitness landscape.

As depicted in Fig. 15.12, we showcase the labels generated by CGP and MAGE. Notably, MAGE successfully separates some closely situated cell clusters that CGP struggles to distinguish, resulting in enhanced accuracy. Additionally, in Fig. 15.13a, we illustrate the evolved graph for CGP, while in Fig. 15.13b, we display the evolved graph for MAGE. Although the utilization of scalar parameters is minimal, involving only two scalar functions, we posit that these dynamic parameters contribute to the observed improvement in generalization and the ability to discern nearby cells. In Table 15.5, we present the AP50 scores achieved by MAGE on both the training and

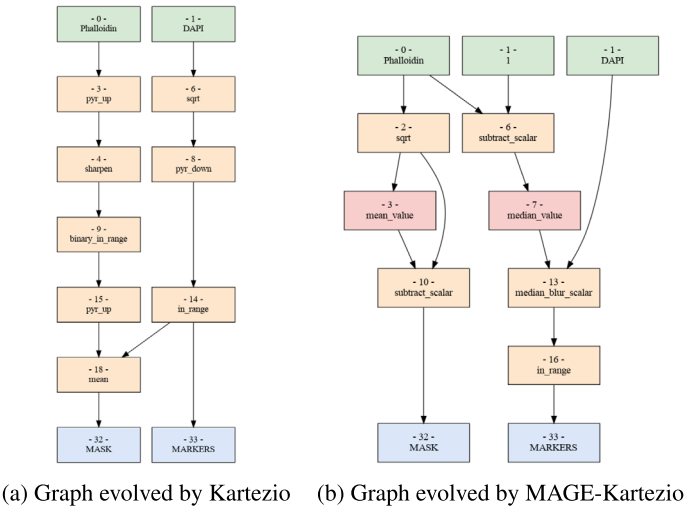


Fig. 15.13 MAGE-Kartezio uses some scalar parameters

Table 15.5 Results on cell image library dataset, a comparison between mono-typed Kartezio and MAGE-Kartezio (10 runs)

	Mean (standard deviation)
Kartezio	0.8161 (0.0547)
MAGE-Kartezio	0.8264 (0.0281)

test datasets. MAGE slightly outperforms CGP across the test data, demonstrating some superior generalization capabilities. Notably, MAGE successfully separates some closely situated cell clusters that CGP struggles to distinguish, resulting in enhanced accuracy.

15.7 Explainable and Interpretable Models

Here we comment on how Kartezio is able to provide explainable and interpretable models, unlike DL methods [9]. As an example, we show a pipeline that obtained the best metric score in the tumor nodules within IHC-stained tissue specimens dataset and comment its behavior, Fig. 15.14. The first step (node 3) of this pipeline consisted of subtracting the Hue channel (node 0) from the Saturation channel (node 1), allowing for the exclusion of pixels corresponding to CD8+ T cells and to other cells belonging to the tumor microenvironment. Then Kartezio used a Gaussian Blur filter (node 4) in order to smooth the previously obtained area. Next, Kartezio chose a Laplacian filter (node 10) to detect the variations in the image obtained in node 4,

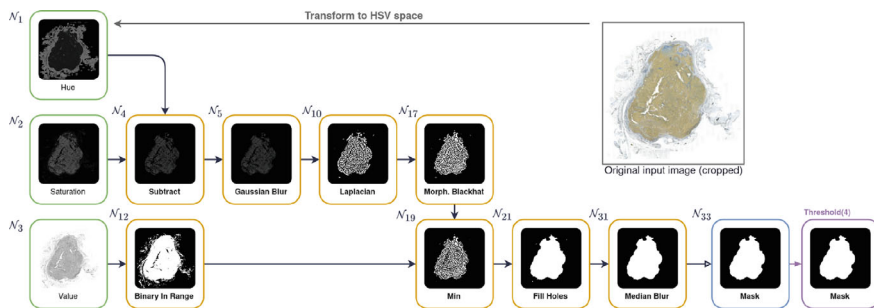


Fig. 15.14 An interpretable pipeline for segmenting tumor nodules within IHC-stained tissue specimens

thus delimiting the tumor borders. Curiously, the sequence Gaussian Blur/Laplacian filters, also known as the Laplacian of Gaussian, is frequently used by image analysts for image segmentation tasks [33]. To highlight the segmented areas, Kartezio used Morphological Black Hat (node 17) which reveals small objects that are darker than their surroundings.

In parallel, node 12 created a binary mask of the Value channel (brightness) allowing for the establishment of a global map of the area of interest. The next step selected by Kartezio consisted of merging the two maps with the min operator (node 19). This allowed for the elimination of all pixels detected by the first calculation (results of node 17) that fell outside the area of interest. Kartezio next selected a Fill Hole filter in order to transform the selected area limits into a solid binary mask. A final step consisted of removing the noise from the edges of the mask using a Median Blur filter (node 31). In its entirety, the described pipeline allowed for the sharp delimitation of tumor nodules with a high degree of accuracy. Crucially, a step-by-step dissection of Kartezio's pipeline, similar to the one we outlined above as an example, can be applied to each pipeline.

15.8 Discussion

For machine learning methods to be adopted in critical decision-making applications, they must be explainable to end users. Interpretability is a way to achieve explainability by using transparent models, where the full functionality of the model can be studied and understood. In this work, we have demonstrated that CGP can evolve interpretable models for biomedical image segmentation through the Kartezio framework. We have shown that Kartezio can evolve models that are competitive with state-of-the-art deep learning algorithms, despite being trained on a limited number of input images. Furthermore, the resultant model of Kartezio is a graph that can be visualized and interpreted by domain experts, providing insights into the model's decision-making process.

A principle challenge in the domain of biomedical image analysis is the scarcity of labeled data. To address this challenge, we have introduced the concept of Active Learning to Kartezio. By dynamically sampling images during evolution, Kartezio can evolve models that are more robust and generalizable. We have shown that this approach can lead to faster convergence and improved performance on the test data. Furthermore, we have demonstrated that Kartezio can be trained with a growing dataset, which can further improve the performance of the algorithm. This study demonstrated the importance of the choice of images for training, motivating future work in sampling methods for symbolic regression.

Finally, the analysis of biomedical images often requires the integration of multiple data types, such as images, clinical data, and genetic data. To address this challenge, we have introduced MAGE-Kartezio, a variant of Kartezio that allows for multimodal data types, such as high-dimensional matrices like images, text, or scalars. We demonstrate that this allows for the dynamic tuning of image function parameters in Kartezio, improving the performance of the algorithm. By allowing the analysis of multiple data types, MAGE-Kartezio opens the door for applications beyond image segmentation, such as classification and other data analysis tasks.

Not all CGP models are interpretable, but our model achieves higher interpretability through its white-box nature and manageable size when compared to deep learning models and future works include how to reinforce interpretability in CGP to develop models that are even more trustful and acceptable. Lastly, the ability of Kartezio to distinguish overlapping cells remains a subject for further study, a subject of importance for events such as mitosis.

References

1. Ahmad, A.M., Khan, G.M., Mahmud, S.A., Miller, J.F.: Breast cancer detection using cartesian genetic programming evolved artificial neural networks. In: Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation, GECCO '12, p. 1031–1038. Association for Computing Machinery, New York, NY, USA (2012). <https://doi.org/10.1145/2330163.2330307>
2. Alavi, A.H., Sadrossadat, E.: New design equations for estimation of ultimate bearing capacity of shallow foundations resting on rock masses. *Geosci. Front.* **7**(1), 91–99 (2016)
3. Ben Hamida, S., Hmida, H., Borgi, A., Rukoz, M.: Adaptive sampling for active learning with genetic programming. *Cognitive Systems Research* **65**, 23–39 (2021) 10.1016/j.cogsys.2020.08.008. URL <https://www.sciencedirect.com/science/article/pii/S1389041720300541>
4. Beucher, S., Meyer, F.: The morphological approach to segmentation: the watershed transformation. In: *Mathematical morphology in image processing*, pp. 433–481. CRC Press (2018)
5. Biau, J., Wilson, D., Cussat-Blanc, S., Luga, H.: Improving image filters with cartesian genetic programming. In: *IJCCI*, pp. 17–27 (2021)
6. Bradski, G.: The opencv library. *Dr. Dobbs's Journal of Software Tools* (2000)
7. Casanova, A., Pinheiro, P.O., Rostamzadeh, N., Pal, C.J.: Reinforced active learning for image segmentation. In: *International Conference on Learning Representations* (2020). URL <https://openreview.net/forum?id=SkG6TNFvr>

8. Clegg, J., Walker, J., Miller, J.: A new crossover technique for cartesian genetic programming. pp. 1580–1587 (2007). <https://doi.org/10.1145/1276958.1277276>
9. Cortacero, K., McKenzie, B., Müller, S., Khazen, R., Lafouresse, F., Corsaut, G., Van Acker, N., Frenois, F.X., Lamant, L., Meyer, N., et al.: Evolutionary design of explainable algorithms for biomedical image segmentation. *Nat. Commun.* **14**(1), 7112 (2023)
10. De La Torre, C., Cortacero, K., Wilson, D.G., Cussat-Blanc, S.: Multimodal adaptive graph evolution. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '24. Association for Computing Machinery, New York, NY, USA* (2024). <https://doi.org/10.1145/3638530.3654347>
11. Deng, S., Zhang, X., Yan, W., Chang, E.I.C., Fan, Y., Lai, M., Xu, Y.: Deep learning in digital pathology image analysis: a survey. *Frontiers of medicine* **14**, 470–487 (2020)
12. Esteva, A., Chou, K., Yeung, S., Naik, N., Madani, A., Mottaghi, A., Liu, Y., Topol, E., Dean, J., Socher, R.: Deep learning-enabled medical computer vision. *NPJ digital medicine* **4**(1), 5 (2021)
13. Gaillouchet, M., Desrosiers, C., Lombaert, H.: Active learning for medical image segmentation with stochastic batches. *Medical Image Analysis* **90**, 102,958 (2023). <https://doi.org/10.1016/j.media.2023.102958>. URL <https://www.sciencedirect.com/science/article/pii/S1361841523002189>
14. Gathercole, C., Ross, P.: Dynamic training subset selection for supervised learning in genetic programming. In: Davidor, Y., Schwefel, H.P., Männer, R. (eds.) *Parallel Problem Solving from Nature – PPSN III*, pp. 312–321. Springer, Berlin Heidelberg, Berlin, Heidelberg (1994)
15. Goldman, B., Punch, W.: Reducing wasted evaluations in cartesian genetic programming. pp. 61–72 (2013). https://doi.org/10.1007/978-3-642-37207-0_6
16. Harding, S.: Evolution of image filters on graphics processor units using cartesian genetic programming. In: *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pp. 1921–1928 (2008). <https://doi.org/10.1109/CEC.2008.4631051>
17. Harding, S., Graziano, V., Leitner, J., Schmidhuber, J.: MT-CGP: mixed type cartesian genetic programming. In: *Proceedings of the 14th annual conference on Genetic and evolutionary computation, GECCO '12*, pp. 751–758. Association for Computing Machinery, New York, NY, USA (2012)
18. Harding, S., Leitner, J., Schmidhuber, J.: Cartesian Genetic Programming for Image Processing, pp. 31–44. Springer New York, New York, NY (2013). https://doi.org/10.1007/978-1-4614-6846-2_3
19. Harding, S., Leitner, J., Schmidhuber, J.: Cartesian Genetic Programming for Image Processing (CGP-IP), pp. 31–44 (2013). https://doi.org/10.1007/978-1-4614-6846-2_3
20. Haut, N., Banzhaf, W., Punch, B.: Active learning improves performance on symbolic regression tasks in stackgp. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '22*, p. 550–553. Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3520304.3528941>
21. Haut, N., Punch, B., Banzhaf, W.: Active learning informs symbolic regression model development in genetic programming. In: *Proceedings of the Companion Conference on Genetic and Evolutionary Computation, GECCO '23 Companion*, p. 587–590. Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3583133.3590577>
22. He, K., Gkioxari, G., Dollár, P., Girshick, R.: Mask r-cnn. In: *Proceedings of the IEEE international conference on computer vision*, pp. 2961–2969 (2017)
23. Hmida, H., Hamida, S.B., Borgi, A., Rukoz, M.: Sampling methods in genetic programming learners from large datasets: A comparative study. In: Angelov, P., Manolopoulos, Y., Iliadis, L., Roy, A., Vellasco, M. (eds.) *Advances in Big Data*, pp. 50–60. Springer International Publishing, Cham (2017)
24. Kalkreuth, R., Rudolph, G., Droschinsky, A.: A new subgraph crossover for cartesian genetic programming. In: *Genetic Programming: 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings 20*, pp. 294–310. Springer (2017)

25. Kirsch, A., van Amersfoort, J., Gal, Y.: Batchbald: Efficient and diverse batch acquisition for deep bayesian active learning. In: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, R. Garnett (eds.) *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc. (2019). URL https://proceedings.neurips.cc/paper_files/paper/2019/file/95323660ed2124450caa2c46b5ed90-Paper.pdf
26. Koza, J.R.: Genetic programming as a means for programming computers by natural selection. *Stat. Comput.* **4**(2), 87–112 (1994)
27. Lavinás, Y., Cortacero, K., Cussat-Blanc, S.: Evolving graphs with cartesian genetic programming with lexibase selection. In: *Proceedings of the Companion Conference on Genetic and Evolutionary Computation, GECCO '23 Companion*, p. 1920–1924. Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3583133.3596402>
28. Lavinás, Y., Haut, N., Punch, W., Banzhaf, W., Cussat-Blanc, S.: Data sampling via active learning in cartesian genetic programming for biomedical data. In: *2024 IEEE Congress on Evolutionary Computation (CEC)* (2024)
29. Lavinás, Y., Haut, N., Punch, W., Banzhaf, W., Cussat-Blanc, S.: Multimodal adaptive graph evolution. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '24*. Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3638530.3654202>
30. Leitner, J., Förster, A., Schmidhuber, J.: Improving robot vision models for object detection through interaction. In: *2014 International Joint Conference on Neural Networks (IJCNN)*, pp. 3355–3362. IEEE (2014)
31. Leitner, J., Harding, S., Frank, M., Förster, A., Schmidhuber, J.: Humanoid learns to detect its own hands (2013). <https://doi.org/10.1109/CEC.2013.6557729>
32. Leitner, J., Harding, S., Förster, A., Schmidhuber, J.: Mars terrain image classification using cartesian genetic programming (2012)
33. Marr, D.C., Hildreth, E.C.: Theory of edge detection. *Proceedings of the Royal Society of London. Series B. Biological Sciences* **207**, 187–217 (1979). URL <https://api.semanticscholar.org/CorpusID:2150419>
34. Martinek, T., Sekanina, L.: An evolvable image filter: Experimental evaluation of a complete hardware implementation in fpga. pp. 76–85 (2005). https://doi.org/10.1007/11549703_8
35. Miller, J.: Cartesian genetic programming: its status and future. *Genetic Programming and Evolvable Machines* **21** (2020). <https://doi.org/10.1007/s10710-019-09360-6>
36. Miller, J.F.: An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2, GECCO'99*, p. 1135–1142. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)
37. Nadizar, G., Medvet, E., Wilson, D.G.: Naturally interpretable control policies via graph-based genetic programming. In: *European Conference on Genetic Programming (Part of EvoStar)*, pp. 73–89. Springer (2024)
38. Nath, V., Yang, D., Landman, B.A., Xu, D., Roth, H.R.: Diminishing uncertainty within the training pool: Active learning for medical image segmentation. *IEEE Trans. Med. Imaging* **40**(10), 2534–2547 (2021). <https://doi.org/10.1109/TMI.2020.3048055>
39. Paris, P.C.D., Pedrino, E.C., Nicoletti, M.: Automatic learning of image filters using cartesian genetic programming. *Integrated Computer-Aided Engineering* **22**(2), 135–151 (2015)
40. Rezatofighi, H., Tsoi, N., Gwak, J., Sadeghian, A., Reid, I., Savarese, S.: Generalized intersection over union: A metric and a loss for bounding box regression. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019)
41. Rudin, C.: Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature machine intelligence* **1**(5), 206–215 (2019)
42. Schmidt, U., Weigert, M., Broaddus, C., Myers, G.: Cell detection with star-convex polygons. In: *Medical Image Computing and Computer Assisted Intervention–MICCAI 2018: 21st International Conference, Granada, Spain, September 16–20, 2018, Proceedings, Part II* **11**, pp. 265–273. Springer (2018)

43. Sekanina, L.: Image filter design with evolvable hardware. pp. 21–27 (2002). https://doi.org/10.1007/3-540-46004-7_26
44. Sekanina, L., Harding, S., Banzhaf, W., Kowaliw, T.: Image processing and CGP **43**, 181–215 (2011). https://doi.org/10.1007/978-3-642-17310-3_6
45. Sekanina, L., Martinek, T.: Evolving image operators directly in hardware. *Eurasip Book Series on Signal Processing and Communications* **8** (2008)
46. Stringer, C., Wang, T., Michaelos, M., Pachitariu, M.: Cellpose: a generalist algorithm for cellular segmentation. *Nat. Methods* **18**(1), 100–106 (2021)
47. Suganuma, M., Shirakawa, S., Nagao, T.: Designing Convolutional Neural Network Architectures Using Cartesian Genetic Programming, pp. 185–208. Springer Singapore, Singapore (2020). https://doi.org/10.1007/978-981-15-3685-4_7
48. De la Torre, C., Cortacero, K., Wilson, D.G., Cussat-Blanc, S.: Multimodal adaptive graph evolution. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '24*, p. To Appear. Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3638530.3654202>
49. Vladislavleva, E., Smits, G., den Hertog, D.: On the importance of data balancing for symbolic regression. *IEEE Trans. Evol. Comput.* **14**(2), 252–277 (2010). <https://doi.org/10.1109/TEVC.2009.2029697>
50. Van der Walt, S., Schönberger, J.L., Nunez-Iglesias, J., Boulogne, F., Warner, J.D., Yager, N., Gouillart, E., Yu, T.: scikit-image: image processing in python. *PeerJ* **2**, e453 (2014)
51. Wilson, D.G., Cussat-Blanc, S., Luga, H., Miller, J.F.: Evolving simple programs for playing atari games. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '18*, p. 229–236. Association for Computing Machinery, New York, NY, USA (2018). <https://doi.org/10.1145/3205455.3205578>

Chapter 16

Crafting Generative Art Through Genetic Improvement: Managing Creative Outputs in Diverse Fitness Landscapes



Erik M. Fredericks, Denton Bobeldyk, and Jared M. Moore

Abstract Generative art is a rules-driven approach to creating artistic outputs in various mediums. For example, a fluid simulation can govern the flow of colored pixels across a digital display or a rectangle placement algorithm can yield a Mondrian-style painting. Previously, we investigated how genetic improvement, a sub-field of genetic programming, can automatically create and optimize generative art drawing programs. One challenge of applying genetic improvement to generative art is defining fitness functions and their interaction in a many-objective evolutionary algorithm such as Lexicase selection. Here, we assess the impact of each fitness function in terms of the their individual effects on generated images, characteristics of generated programs, and impact of bloat on this specific domain. Furthermore, we have added an additional fitness function that uses a classifier for mimicking a human's assessment as to whether an output is considered as "art." This classifier is trained on a dataset of input images resembling the glitch art aesthetic that we aim to create. Our experimental results show that with few fitness functions, individual generative techniques sweep across populations. Moreover, we found that compositions tended to be driven by one technique with our current fitness functions. Lastly, we show that our classifier is best suited for filtering out noisy images, ideally leading toward more outputs relevant to user preference.

Keywords Generative art · Grammatical evolution · Genetic improvement · Genetic programming · Image analysis · Fitness comparison · Convolutional neural network · Classifier · Aesthetic assessment · Image optimization

E. M. Fredericks (✉) · D. Bobeldyk · J. M. Moore
College of Computing, Grand Valley State University, Allendale, MI, USA
e-mail: frederer@gvsu.edu

D. Bobeldyk
e-mail: bobeldyd@gvsu.edu

J. M. Moore
e-mail: moorejarm@gvsu.edu

16.1 Introduction

Generative art produces an output using a rules-based approach, algorithm, or other heuristic [4, 7, 17–19, 33, 50]. Example outputs include digital images/animations, models intended for 3D printing, and vector images intended for pen plotting. It can be used to teach programming concepts to students [3, 15, 21, 42, 48], visualize trends in datasets [10], or illustrate the behaviors of complex algorithms [4, 7, 17, 33, 50]. We follow the definition of generative art proposed by Boden and Edmunds [4], however we acknowledge that alternative definitions exist. Generative art is highly amenable to genetic programming (GP) as it can comprise one or more drawing functions that result in substantially different outputs, depending on the parameterization. For this research, we use grammatical evolution (GE), a subset of GP, for evolving programs using an input grammar constraining the solution space.

Computer-based generative art may rely upon one or more parameterized drawing functions to create an output, where function parameters are used to induce variety in the output each time the function is invoked [4, 7, 17, 21, 48]. For example, the behavior and output of a flow field (c.f., Sect. 16.2) may be varied by changing how the underlying vector field is defined (e.g., octaves and falloff in the case of a Perlin noise-based field [43]), whether the drawing mechanism will be connected lines or individual particles, and the color palette used when drawing. Additionally, the digital representation of the flow field could be realized as a physical artistic piece by using a pen plotter or 3D printer. Each technique and associated parameters can be encoded within a genome and result in a drawing program that can be evolved with GP. However, assessing the “goodness” of a solution is a non-trivial problem as preference is difficult to mathematically quantify and is individually subjective. We use GE for constraining the structure of evolved programs with a grammar [47, 50]. This approach can be considered as a form of genetic improvement (GI) or automatically improving programs using evolution [28].

Assessing the individual and combined impact of the fitness functions in an open-ended, many-objective evolutionary process is an ongoing and challenging problem. Our prior research efforts (*GenerativeGI*) aimed to create outputs that resemble glitchy or corrupted images for aesthetic purposes [18, 19]. However, automatically assessing the quality of those outputs is non-trivial and determining if a generated image is “art” is subjective, necessitating a human examiner. Further complicating assessments are the different interpretations of computational aesthetics as noted by Greenfield [22]. Our primary goal with *GenerativeGI* is to automate the assessment process suitable for GP. As human preference is difficult to mathematically quantify, we use proxy functions (e.g., maximizing the novelty of the generated program, incorporating as many techniques as possible, maximizing pixel difference

between outputs, etc.) to attempt to emulate the preferences of the designer. As the number of fitness functions grows to incorporate various dimensions of artistic evaluation, the individual influence of each fitness function is not immediately evident. This chapter investigates the impact of the fitness objectives on the overall evolutionary process when creating generative art.

We perform a combinatorial analysis of fitness objectives within the evolutionary selection process to determine the impact and contribution of each to the overall outputs. The generated programs and outputs are evaluated in terms of their relevance to the desired aesthetic style of outputs, fitness values, and execution metrics. Additionally, we include a new fitness function that uses a machine learning classifier, trained on an abstract art corpus [5], to both provide additional evolutionary pressure and serve as a post-processing mechanism for an “art vs. not art” evaluation. Our experimental results indicate that fewer fitness functions result in generative art techniques sweeping across populations, leading to very similar outputs. However, we also note that with all of our fitness functions included that we still see sweeping occurring, indicating that other fitness functions, evolutionary pressures, or diversity preservation options are required for further novelty. Additionally, we found that *GenerativeGI* tended to evolve solutions that were skewed toward a dominant generative technique per experimental configuration. We also found, as is somewhat expected, that program bloat was occurring in our outputs as well. Bloat is potentially useful in this domain, although it also can be a challenge. Additional drawing functions can lead to interesting/unexpected outputs at the expense of the time required to render the techniques. Lastly, we found that our classifier tends to classify “noisy” outputs as “not art,” ideally leading to a better evolutionary feedback mechanism for moving away from glitch art. For archival purposes, we include our full datasets on Zenodo¹ and updated codebase on Github.²

The remainder of this chapter is organized as follows. Section 16.2 overviews generative art, GE and GI, our classifier, and related work. Section 16.3 then presents and discusses our investigation into fitness function impact on *GenerativeGI*. Finally, Sect. 16.4 summarizes our results and presents future directions for research.

16.2 Generative Art, Genetic Improvement, and Convolutional Neural Network Classifiers

This section provides relevant information on generative art, GE and GI, and our convolutional neural network (CNN) classifier. We additionally present related work in this section.

¹ See <https://zenodo.org/records/12796744>.

² See <https://github.com/efredericks/GenerativeGI/tree/GPTP-args>.

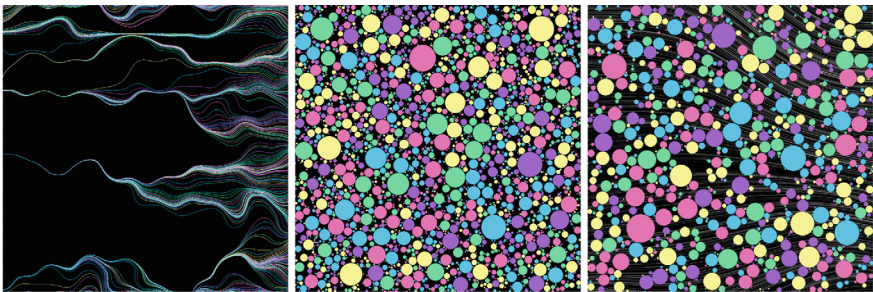
16.2.1 Generative Art

Generative art is typically considered to be a process for creating an artistic output (either digitally or physically) via a heuristic or algorithm [4].³ Famous generative artists include Joshua Davis, Zack Lieberman, Vera Molnár, Frieder Nake, Casey Reas, Manfred Mohr, Lillian Schwartz, Georg Nees, Roman Verostko, Rafaël Rozendaal, and Sol LeWitt. Moreover, generative art can be considered as an umbrella term for multiple forms of creative outputs, including artistic expression, visualizations of datasets or algorithms, and creative coding purposes for teaching programming concepts [4, 7, 10, 17–19, 33, 50]. Any form of visualization can serve as the basis of a generative art program. We briefly introduce two examples to illustrate the process: flow fields and circle packing.⁴

Flow fields visualize motion along vector fields, typically in terms of velocity or orientation [7]. Figure 16.1a presents a sample flow field output using particle-driven motion across a Perlin noise-mapped vector field [43]. In this output, a noise value (using Perlin noise) is calculated based on x - and y -coordinates within the image canvas and that value is then mapped to an angle on $[-2.0 * \pi, 2.0 * \pi]$. A particle (comprising a position and color) will draw a point on the canvas with its specified color at its current position. The particle will then sample the underlying noise field and calculate its next position based on the corresponding angle.

Circle packing aims to fit as many circles as possible into a given space while requiring that no circle overlaps with another. Figure 16.1b presents a sample circle packing output.

While each algorithm alone can create interesting artistic outputs, the complexity of the output can significantly increase when techniques are combined. Figure 16.1c



(a) Flow field example. (b) Circle packing exam- (c) Combined flow field /
ple. circle packing example.

Fig. 16.1 Sample generative art outputs

³ This differs significantly from the recent explosion of interest in outputs created via generative adversarial networks, though the terms are similar.

⁴ A larger number of examples included within GenerativeGI are described in detail in [19].

demonstrates the output when a flow field is run and then followed by circle packing (with a different color palette for differentiation).

A combination of artistic techniques constitutes a drawing program where the invocation and ordering of art functions, along with their respective parameter instantiation(s), varies from execution to execution. Such a process is amenable to evolutionary computation for optimization in selection of the drawing techniques, the configuration of parameters for each, and the order in which they are executed [2, 49].

Note that evolutionary computation has been used to support numerous creative endeavors, including [2, 8, 9, 11, 12, 24, 25, 34, 35, 49, 50]. An index of papers published at the Conference on Computational Intelligence in Music, Sound, Art and Design (EvoMUSART) from 2003 onward provides a comprehensive list of highly related papers to this domain [36].

In our prior work, *GenerativeGI*, we used GE as the means for evolving drawing programs within a constrained solution space [18, 19]. Genomes within *GenerativeGI* are defined as a combination of parameterized drawing techniques expressed as a string, where we used single-point crossover and single-point mutation. Mutation will either select a new drawing technique with new parameters at the mutation point, update parameters for the existing technique at the mutation point, or shuffle the order of all individual techniques within the genome when activated.

Large-language model-based techniques may leverage GPT-3 [16] and GPT-4 [40, 46] to support synthesis using massive datasets of existing samples. Such an approach is typically governed by user prompts for creating relevant outputs, with techniques including diffusion models [52] and generative adversarial networks (GANs) [20]. Conversely, our approach relies on developer-provided drawing techniques and evolutionary search to discover ideal drawing programs. Of note is that there are significant concerns regarding image copyright and ownership for GAN/diffusion model training datasets [1].

16.2.2 Grammatical Evolution and Genetic Improvement

GE is a form of GP in that the genome is represented as a variable-length byte list and parsed by a grammar (typically in Backus–Naur form), as opposed to a tree or stack-based representation [26, 47, 50]. This genome places additional constraints on generated outputs as the only options that are feasible are syntactically correct to the grammar, ideally reducing bloat and/or invalid programs. However, such an approach requires a significant amount of domain expertise to properly construct the grammar.

GI extends the concepts of GP/GE by improving the source code and/or parameters of a generated program [23, 29, 30, 44, 45]. Examples of GI include automated bug repair via software patches [31], repurposing old functions for new program uses [30], and reducing program energy consumption via optimized code processes [6].

GenerativeGI [18, 19] uses GE and GI to automatically create parameterized sequences of drawing functions that yield artistic outputs, where our outputs are governed by a many-objective selection mechanism (i.e., Lexicase selection [51]) to enable search in a large solution space. Here, selection of individuals for crossover/mutation occurs by comparing individual fitness objectives in a randomized order. If an individual performs better in the compared objective then it will be immediately selected for reproduction. If a tie occurs then the selection process moves on to the next objective. GenerativeGI uses both integer and real-valued objectives. Comparison in real-valued objectives is enabled by ϵ -Lexicase selection for ranking individuals within a specified range [27], where our specified range is set to $\epsilon = 0.85$ based upon empirical evidence.

Related to Lexicase selection are multi-objective optimization approaches, such as NSGA-II [14]. While pareto-based approaches are powerful for multiple objectives, we intend to continue adding new fitness functions to the point where pareto-based approaches may begin to break down.

Regardless, fitness functions are typically defined with some objective in mind. There seems to be little existing work on teasing apart the impact of individual fitness functions on the evolutionary process as a whole, instead relying on domain expertise to determine if a fitness function is “good.” Previously, Panait and Luke investigated how competitive fitness (i.e., one- or multi-population individuals working cooperatively or competitively toward a common goal, similar to co-evolution) can impact evolutionary outcomes [41]. Such an approach can be an interesting future direction for GenerativeGI, as currently it makes use of traditional, single-population individuals. Deb and Gupta previously investigated knee points within Pareto fronts in multi-objective optimization as points of potential trade-off optimization [13]. In terms of our research, knee points may indicate similar tradeoffs for our fitness functions, however they do not necessarily indicate the usefulness of the fitness function itself.

16.2.3 *Art Classification*

Human preference is challenging to mathematically quantify, yet there was a critical need to include a technique for measuring the extent to which an image resembles “art” that would facilitate gradual improvements in successive iterations toward the “art” category. Integrating a human reviewer into the evolutionary loop would be infeasible as each of the reviews would be time-consuming and add an exponential amount of time to each run. As such, machine learning could provide a proxy for human preference when trained on a relevant dataset.

A CNN is a type of machine learning model that is capable of learning and then predicting to what class an image belongs [32]. The model is trained on images that belong to each class and learns how to extract relevant features that are used for class prediction. Output is typically a numeric value associated with how confident the model feels an image belongs to a certain class, where the class that has the largest

numeric value is selected by the model. To integrate this model properly into our pipeline, the feature extractor component was separated from the classifier, where generated images were then analyzed to detect if they contain features found by the model.

A standard CNN model (see Fig. 16.2a) typically contains a configurable amount of layers, where the first layers are responsible for extracting features and the latter layers are responsible for classifying learned features. The classification layers generally take a high-dimensional vector and reduce those layers to a smaller dimensional vector (i.e., the output layer). Our CNN implementation removes the typical classification layers to generate a vector of learned features (see Fig. 16.2b) for use within the evolutionary loop. The generated feature vector can measure how close an image is to either the “art” or “not art” feature space.

We used a dataset of abstract art [5] and randomly generated noise images to determine the “art” and “not art” feature spaces, respectively. A standard CNN model was trained on the dataset in an attempt to learn the features of the “art” class. The classification layers were then removed after training and each of the art images were applied as input to the model. The mean of the output feature vectors generated from all of the “art” class images was calculated as well as those from “not art.” The revised model accepts a new input image and outputs a fitness score that could be used to determine how close the image was to the “art” class.

A potential challenge is that classifiers have been proven to be easily fooled into mislabeling outputs in both artistic endeavors and other classification tasks [11, 12, 34, 38]. Such results indicate that classifiers are not always generalizable and such must be recognized when including a classifier as part of the feedback process. For our purposes, we use our CNN as a companion objective within the selection process and do not fully rely on it for quantification. As is described in Sect. 16.3, our classifier is helpful in reducing noisy or blurry outputs and can help the designer focus on specific images within a large number of outputs.

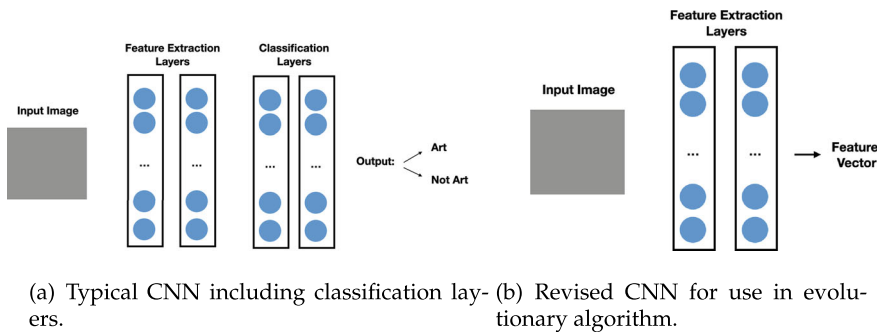


Fig. 16.2: CNN visualizations.

Fig. 16.2 CNN visualizations

16.3 Lexicase Selection, Fitness Landscapes, and Examining Search Criteria

Generally, fitness functions are defined with developer domain knowledge and an aim to optimize a specific aspect of a possible solution. Depending on the domain, crafting fitness functions can be straightforward, translating standard problem metrics into a fitness objective. For example, a fitness function for a robot learning to walk might maximize absolute distance traveled from its origin. When moving to multiple objectives, however, Pareto fronts and competing objectives [13, 14] can introduce complexity, obscuring individual fitness function contributions. The impact and usefulness of fitness functions can be non-trivial to assess when the problem domain is open ended or domain knowledge is lacking. For example, an optimal solution in a generative art application might be guided by any number of fitness functions as each output can induce a different “feature” crafting a desirable composition. Such features could include favoring brush strokes over pixel placement, maximizing negative space in the canvas to provide “breathing room” for drawings, or maintaining a consistent color palette for aesthetic purposes.

In our prior efforts [18, 19], we included fitness functions that were intended to induce a “glitch art” effect (i.e., in the style of aesthetically broken or corrupted images). We introduced a number of fitness functions (c.f., Table 16.1) that guided search toward glitch art while attempting to maximize diversity in the population of images. However, these techniques often resulted in images that appeared noisy and cluttered. Our application domain is fundamentally a problem of human preference that must be mathematically quantified, as incorporating a human in the loop would be impractical given that full evaluations can span multiple days of processing time within a high-performance computing (HPC) environment. To this end, our fitness functions served as proxies for human preference. *GenerativeGI* included the fitness functions as described within Table 16.1, except FF_{ac} , which is a new component introduced for this chapter.

Experiments. In this work, we performed a combinatorial analysis of our fitness objectives through a “leave x out” approach to determine the relative impact of each

Table 16.1 Fitness functions used by *GenerativeGI*. FF_{pc} through FF_{ns} were included in [19]

FF_{pc}	(Maximize) Pairwise comparison (RMS pixel difference between images)
FF_{gc}	(Minimize) Number of duplicate genes
FF_{ut}	(Maximize) Diversity of unique techniques
FF_{cd}	(Maximize) Chebyshev difference between images
FF_{ns}	(Minimize) Negative space (target 70%)
FF_{ac}	(Minimize) Art classifier (numerical “art” or “not art”) <i>Lower values indicate closer relation to “art”-classified images</i>

objective. Depending on the experimental configuration, objectives were toggled on or off during the selection process, however each was still measured during evaluation for post-run processing. We denote the fitness objectives actively used in selection in each experimental configuration (EC) with subscripts. For example, a configuration with negative space (ns) and the art classifier (ac) active is denoted as $EC_{ns,ac}$.

15 replicates per experimental configuration were run. Each replicate comprised 100 individuals evolving for 100 generations. Crossover and mutation rate were set to 0.5 and 0.4, respectively. An individual is created via single-point crossover with genetic material from two parents with a 50% chance, otherwise it is cloned. Mutation is then performed on an individual with a 40% chance using single-point mutation.

Time Analysis. We first evaluated individual technique timing prior to our investigation into fitness as each of our submitted HPC jobs would fail with a wall time error (a maximum of 88 h was specified for each individual replicate). We then set a time limit of three minutes per evaluation for each individual, after which evaluation would cease regardless of genome length. This change enabled all experimental configurations and replicates to complete within a significantly shorter amount of time.

The issue with timeouts can be seen as an effect of program bloat as our genome length was quite large and each drawing technique requires time to execute. Therefore, we analyzed the times of each of our generative art techniques individually. As can be seen from Fig. 16.3, one particular technique (`flow-field`) takes significantly longer than other techniques to run. In reviewing grammar outputs from cancelled runs, we noted that the `flow-field` was highly prevalent in the population with some individuals having multiple flow fields in their genome. As we have a second version of the flow field algorithm (i.e., `flow-field-2`), we opted to remove `flow-field` and re-run our experiments with an increased timeout of four minutes per evaluation.

Generative art techniques can be computationally expensive requiring calculation at the individual pixel level or evaluating a high number of individual agents used to create an artifact. Combining multiple techniques together can yield significant increases in runtime that make evolutionary processes prohibitively expensive from a wall clock perspective. Still, maintaining a variety of techniques in an individual's genome enhances the possibility of a novel combination of genes to yield "interesting" results. To balance these competing challenges, we implemented a maximum evaluation time (three minutes) for an individual to express its genome in an image and re-ran our experimental replicates for analysis. If a composition was still being generated at three minutes, the currently active technique was allowed to complete and the remaining genes were not expressed.

Next, we visually inspected the outputs from the final generation of each experimental configuration by generating an image collage. One of the most striking observations was the amount of techniques sweeping across populations, depending on the number of fitness functions included. For example, Fig. 16.4a shows a collage of outputs from $EC_{ut,ac}$ (i.e., FF_{ut} and FF_{ac}). Even though a fitness objective was activated to maximize the number of unique techniques, visually it appears that

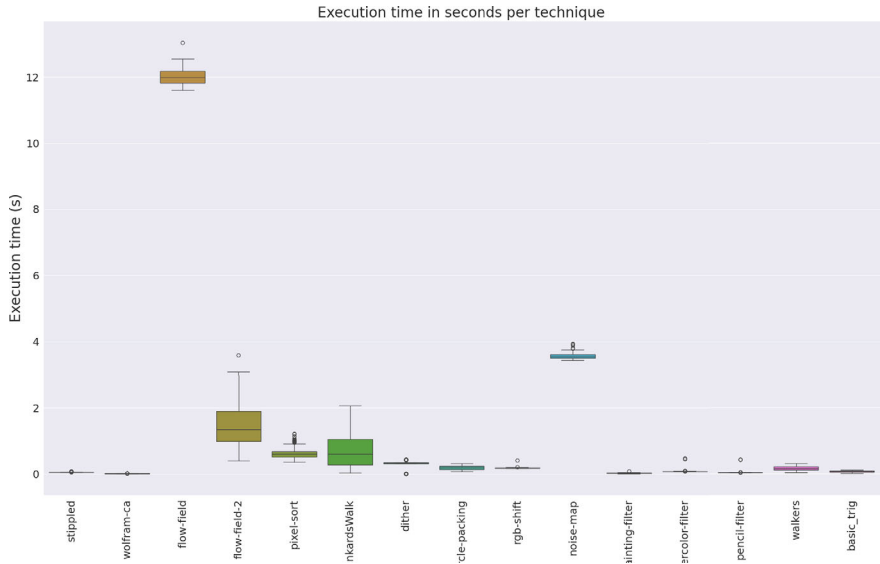


Fig. 16.3 Total time required for each individual technique across 100 randomly instantiated invocations

the population converges on a common technique (i.e., `basic-trig`, or drawing trigonometric functions) as well as a common color palette. Whereas outputs from the initial population from this configuration were quite diverse.

Conversely, Fig. 16.4b demonstrates a run with more fitness objectives active during selection. Techniques are still clearly sweeping across the population, however

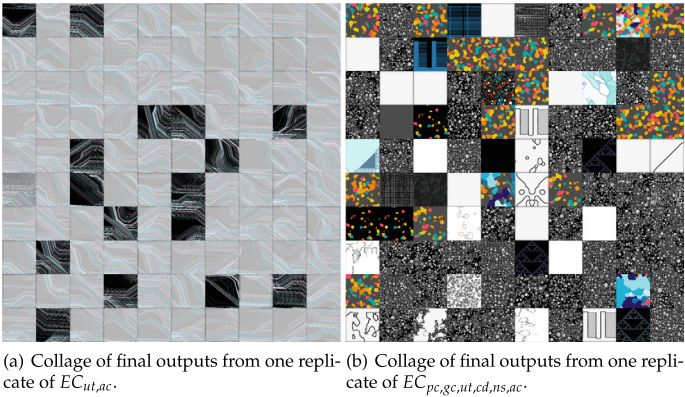


Fig. 16.4 Sample collages of final outputs from $EC_{ut,ac}$ and $EC_{pc,gc,ut,cd}$ from one replicate. Each collage comprises the outputs from the final evolved population to demonstrate the difference between the different objectives used for selection

there exists more variety in the outputs. This difference is observed across replicates suggesting that activating more fitness objectives may lead to niching in the population.

We then examined the fitness values across each configuration. Although only a subset of fitness functions were active in selection, all were calculated per individual for post-run analysis. Figure 16.5 shows a heatmap comprising each experimental configuration and fitness objectives. Each fitness function returns values on varying ranges, and as such, we have normalized each to a range of [0.0, 1.0], where 1.0 is considered most fit and 0.0 is considered least fit.

All experimental configurations minimize the number of duplicate genes shown by the dark red for the FF_{gc} row. However, many of the other fitness functions are maximized only when they are used in active selection for specific experimental configurations. This behavior is shown in the geometric patterns that emerge horizontally for each FF row. Of note, FF_{ac} remains relatively high across runs suggesting that most experimental configurations produce “art” as determined by the CNN classifier (and colloquially by the authors). Negative space, another common concept in composition, only reaches high scores (near 70% in an image) when actively selected for. While the classifier may be indicating “art” this suggests that negative space is

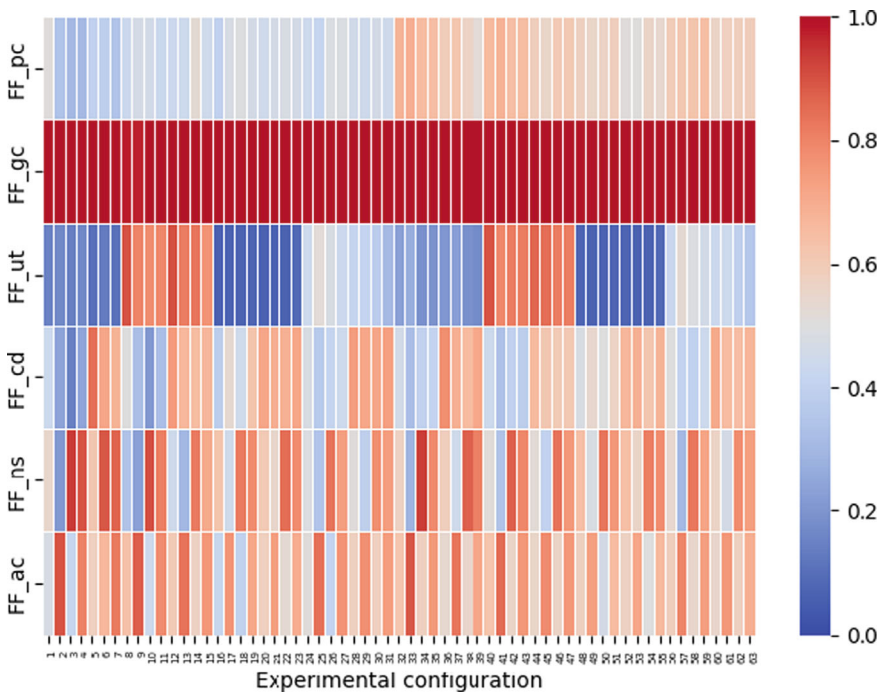


Fig. 16.5 Heatmap of normalized and averaged fitness values for final population per experimental configuration (three-minute evaluation timeout, flow-field removed). Experimental configuration numbers correlate to a truth table of activated fitness objectives

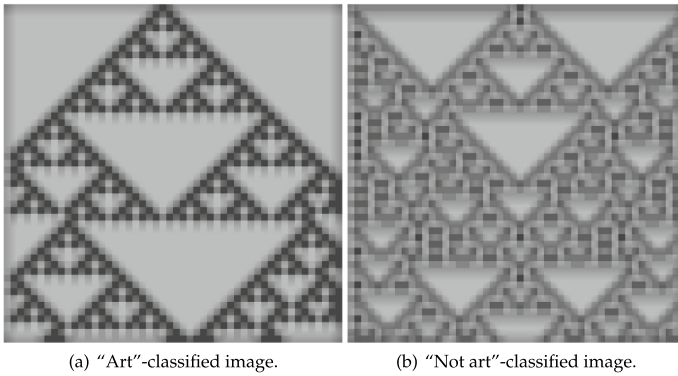


Fig. 16.6 Classified “art” vs. “not art” images from $EC_{pc,gc,ut,cd}$

not being strongly considered by the FF_{ac} classifier as FF_{ns} occasionally has a low score.

Output Classification. In our prior work, the evolved compositions often resembled “glitch art,” our target style. However, they exhibited artifacts such as blurriness or noise not seen in human-crafted glitch art. One possible reason for this is that each fitness function is assessing specific aspects of a composition such as duplicate genes, a target negative space, or maximizing difference between images in a population. Considering the whole of an image in a composition is not amenable for these fitness functions. Here, we added the previously mentioned CNN classifier as FF_{ac} to try and assess images more holistically. After conclusion of the runs, we ran our classifier on the final population of images for all experimental configurations to determine the total number of generated images classified as most resembling our training dataset. This analysis serves two purposes: to ensure that FF_{ac} is functioning as expected within our evolutionary process and to provide the authors with additional information for deeper visual inspection on particular images. For example, an image classified as “not art” may look similar to others in a collage view, however zoomed in they tend to look noisy, cluttered, or blurry. Figure 16.6a shows an example of an output image classified as “art” and Fig. 16.6b shows an example of a “not art” image. As can be seen, the images are visually similar and yet the “not art” image is significantly blurrier, leading to the supposition that the classifier is providing feedback on the validity of generated images. For all experimental configurations, the number of images classified as “art” significantly outnumbers the number classified as “not art.” These results suggest that further refinements/additions to our classifier are necessary to aid the search process.

16.4 Discussion

This chapter has presented an empirical analysis of the impact of fitness objectives within the context of an open-ended optimization problem where human preference supersedes quantitative metrics. Specifically, we investigated how different combinations of fitness functions can influence generative art outputs, given that our defined fitness functions were used in an ad hoc fashion [18, 19]. For this particular domain, results suggest that the number of included fitness objectives during selection can have an impact on artistic techniques sweeping across the population. Moreover, some fitness objectives (i.e., FF_{gc} , FF_{ac}) tended to score high throughout all configurations whereas some fitness objectives (i.e., FF_{ns}) only scored high when active. Finally, we found that our classifier is well suited for filtering images that do not meet basic qualifications for being considered “art” (e.g., noisy or blurry images).

Future directions for this path of research include investigating different selection mechanisms (e.g., Dalex [39], MAP-ELITES [37]) to focus on diversity of results, improving the underlying data structure managing our generative art techniques to enable interactions between individual techniques, and investigating different machine learning classifiers for assessing output quality.

Acknowledgements The authors gratefully acknowledge the reviewers’ comments which helped to improve the manuscript. We gratefully acknowledge support from the Michigan Space Grant Consortium (award number 80NSSC20M0124), resources from the Grand Valley State University High Performance Computing cluster (which is supported by the Academic Research Computing department), and Grand Valley State University. The authors additionally thank Alexander Lalejini and Abigail Diller for their inputs as well.

References

1. Balan, K., Agarwal, S., Jenni, S., Parsons, A., Gilbert, A., Collomosse, J.: Ekila: synthetic media provenance and attribution for generative art. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 913–922 (2023)
2. Baluja, S., Pomerleau, D., Jochem, T.: Towards automated artificial evolution for computer-generated images. *Connect. Sci.* **6**(2–3), 325–354 (1994)
3. Bergstrom, I., Lotto, R.B.: Code bending: a new creative coding practice. *Leonardo* **48**(1), 25–31 (2015)
4. Boden, M.A., Edmonds, E.A.: What is generative art? *Digit. Creat.* **20**(1–2), 21–46 (2009)
5. Boulé, B.: Abstract art gallery (2022). <https://www.kaggle.com/datasets/bryanb/abstract-art-gallery/>. Accessed 13 May 2024
6. Bruce, B.R., Petke, J., Harman, M.: Reducing energy consumption using genetic improvement. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, pp. 1327–1334 (2015)
7. Cabral, B., Leedom, L.C.: Imaging vector fields using line integral convolution. In: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, pp. 263–270 (1993)
8. Clune, J., Lipson, H.: Evolving three-dimensional objects with a generative encoding inspired by developmental biology. In: ECAL, pp. 141–148 (2011)

9. Clune, J., Yosinski, J., Doan, E., Lipson, H.: Endlessforms.com: collaboratively evolving objects and 3d printing them
10. Comba, J.L.: Data visualization for the understanding of covid-19. *Comput. Sci. Eng.* **22**(6), 81–86 (2020)
11. Correia, J., Machado, P., Romero, J., Carballal, A.: Evolving figurative images using expression-based evolutionary art. In: *Iccc*, pp. 24–31 (2013)
12. Correia, J., Machado, P., Romero, J., Martins, P., Amílcar Cardoso, F.: Breaking the Mould an Evolutionary Quest for Innovation Through Style Change, pp. 353–398. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-319-43610-4_16
13. Deb, K., Gupta, S.: Understanding knee points in bicriteria problems and their implications as preferred solution principles. *Eng. Optim.* **43**(11), 1175–1204 (2011)
14. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.* **6**(2), 182–197 (2002)
15. Dehouche, N., Dehouche, K.: What is in a text-to-image prompt: the potential of stable diffusion in visual arts education (2023). [arXiv:2301.01902](https://arxiv.org/abs/2301.01902)
16. Floridi, L., Chiriatti, M.: GPT-3: its nature, scope, limits, and consequences. *Minds Mach.* **30**(4), 681–694 (2020)
17. Forbes, A.G., Höllerer, T., Legrady, G.: Generative fluid profiles for interactive media arts projects. In: *Proceedings of the Symposium on Computational Aesthetics*, pp. 37–43 (2013)
18. Fredericks, E.M., Diller, A.C., Moore, J.M.: Generative art via grammatical evolution. In: *Proceeding of the 12th International Workshop on Genetic Improvement*
19. Fredericks, E.M., Moore, J.M., Diller, A.C.: GenerativeGI: creating generative art with genetic improvement. *Autom. Softw. Eng.* **31**(1), 23 (2024)
20. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial networks. *Commun. ACM* **63**(11), 139–144 (2020)
21. Greenberg, I.: *Processing: Creative Coding and Computational Art*. Apress, Berkeley, CA (2007)
22. Greenfield, G.: On the origins of the term “computational aesthetics”. In: *Computational Aesthetics in Graphics, Visualization and Imaging*, pp. 9–12 (2005)
23. Harman, M., Petke, J.: Gi4gi: improving genetic improvement fitness functions. In: *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pp. 793–794 (2015)
24. den Heijer, E.: Evolving glitch art. In: Machado, P., McDermott, J., Carballal, A. (eds.) *Evolutionary and Biologically Inspired Music, Sound, Art and Design*, pp. 109–120. Springer, Berlin, Heidelberg (2013)
25. Johnson, C.G.: Fitness in evolutionary art and music: a taxonomy and future prospects. *Int. J. Arts Technol.* **9**(1), 4–25 (2016)
26. Koza, J.R.: Genetic programming as a means for programming computers by natural selection. *Stat. Comput.* **4**(2), 87–112 (1994)
27. La Cava, W., Spector, L., Danai, K.: Epsilon-lexicase selection for regression. In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016, GECCO '16*, pp. 741–748. Association for Computing Machinery, New York, NY, USA (2016). <https://doi.org/10.1145/2908812.2908898>
28. Langdon, W.B.: Genetic improvement of genetic programming. In: *2020 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8. IEEE (2020)
29. Langdon, W.B., Harman, M.: Optimizing existing software with genetic programming. *IEEE Trans. Evol. Comput.* **19**(1), 118–135 (2014)
30. Langdon, W.B., Lam, B.Y.H., Modat, M., Petke, J., Harman, M.: Genetic improvement of GPU software. *Genet. Program. Evolvable Mach.* **18**, 5–44 (2017)
31. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: fixing 55 out of 105 bugs for \$8 each. In: *2012 34th International Conference on Software Engineering (ICSE)*, pp. 3–13. IEEE (2012)
32. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. *Nature* **521**(7553), 436–444 (2015)

33. Liu, H., Liu, X.: Generative art images by complex functions based genetic algorithm. In: Trends in Computer Aided Innovation, pp. 125–134 (2007)
34. Machado, P., Correia, J., Romero, J.: Improving face detection. In: Genetic Programming: 15th European Conference, EuroGP 2012, Málaga, Spain, April 11–13, 2012. Proceedings 15, pp. 73–84. Springer (2012)
35. Machado, P., Romero, J., Manaris, B.: Experiments in computational aesthetics: an iterative approach to stylistic change in evolutionary art. In: The Art of Artificial Evolution: A Handbook on Evolutionary Art and Music, pp. 381–415. Springer (2008)
36. Martins, T., Correia, J.: Evomusart index (2024). <https://evomusart-index.dei.uc.pt/info>. Accessed 09 Jul 2024
37. Mouret, J.B., Clune, J.: Illuminating search spaces by mapping elites (2015)
38. Nguyen, A., Yosinski, J., Clune, J.: Deep neural networks are easily fooled: high confidence predictions for unrecognizable images. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 427–436 (2015)
39. Ni, A., Ding, L., Spector, L.: Dalex: Lexicase-like selection via diverse aggregation. In: European Conference on Genetic Programming (Part of EvoStar), pp. 90–107. Springer (2024)
40. OpenAI: Gpt-4 technical report (2023)
41. Panait, L., Luke, S.: A comparison of two competitive fitness functions. In: Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation, pp. 503–511 (2002)
42. Peppler, K., Kafai, Y.: Creative coding: programming for personal expression **30**(2008), 314 (2005)
43. Perlin, K.: Noise hardware. in real-time shading. In: SIGGRAPH 2002 Course Notes, pp. 1–26 (2001)
44. Petke, J., Haraldsson, S.O., Harman, M., Langdon, W.B., White, D.R., Woodward, J.R.: Genetic improvement of software: a comprehensive survey. IEEE Trans. Evol. Comput. **22**(3), 415–432 (2017)
45. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement and code transplants to specialise a c++ program to a problem class. In: Genetic Programming: 17th European Conference, EuroGP 2014, Granada, Spain, April 23–25, 2014, Revised Selected Papers 17, pp. 137–149. Springer (2014)
46. Phung, T., Pădurean, V.A., Cambronero, J., Gulwani, S., Kohn, T., Majumdar, R., Singla, A., Soares, G.: Generative AI for programming education: Benchmarking chatgpt, gpt-4, and human tutors. Int. J. Manag. **21**(2), 100,790 (2023)
47. Ryan, C., Collins, J.J., Neill, M.O.: Grammatical evolution: evolving programs for an arbitrary language. In: Genetic Programming: First European Workshop, EuroGP’98 Paris, France, April 14–15, 1998 Proceedings 1, pp. 83–96. Springer (1998)
48. Shiffman, D., Fry, S., Marsh, Z.: The nature of code. D. Shiffman (2012)
49. Sims, K.: Interactive evolution of equations for procedural models. Int. J. Manag. **9**, 466–476 (1993)
50. Smedt, T.D., Lechat, L., Daelemans, W.: Generative art inspired by nature, using nodebox. In: European Conference on the Applications of Evolutionary Computation, pp. 264–272. Springer (2011)
51. Spector, L.: Assessment of problem modality by differential performance of Lexicase selection in genetic programming: a preliminary report. In: Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation, pp. 401–408. ACM, Philadelphia, Pennsylvania, USA (2012)
52. Zhang, C., Zhang, C., Zhang, M., Kweon, I.S.: Text-to-image diffusion model in generative AI: a survey (2023). [arXiv:2303.07909](https://arxiv.org/abs/2303.07909)

Chapter 17

Cell Regulation and the Early Evolution of Autonomous Control



Anselmo C. Pontes and Nicholas L. Panchy

Abstract Biological systems exhibit remarkable complexity, with modern multicellular organisms showcasing intricate behaviors and intelligence. However, this complexity is relatively recent. For the vast majority of natural history, only single-celled organisms existed, and it was within these seemingly simple systems that evolution produced the functions and behaviors fundamental to life as we know it. Many of these functions are regulatory and can be viewed as control systems. Understanding their evolution in single cells is key to explaining how these simple organisms gave rise to more integrated life forms. To address this, we introduce Elfa, a digital evolution platform that emulates the genome and regulatory systems of prokaryotic cells while abstracting complex cellular phenotypes. Designed both as a model for studying cellular evolution and as a tool for solving practical problems, Elfa provides a biologically inspired framework for exploring evolutionary dynamics. In our initial experiments, simulating photosynthetic bacteria in variable light environments, we show that regulation evolves as expected in response to environmental variation, but the rate of fixation of regulation increases with the combination of different sources of environmental variation. Most promising, though, is the degree of evolvability demonstrated by our model in this challenging multi-objective optimization problem, where the traits under evolution are highly epistatic and the dynamics to be regulated are noisy and mostly non-linear. These findings demonstrate that Elfa is an effective model for studying biological evolution and for exploring how environmental factors drive the evolution of regulatory mechanisms in early cells.

A. C. Pontes (✉)

Autogenetics Research Lab, Washington DC, USA

e-mail: anselmo@autogenetics.ai

N. L. Panchy

Institute for Cyber Enabled Research, Michigan State University, East Lansing, USA

e-mail: panchyni@msu.edu

17.1 Introduction

Autonomy is perhaps life's most underrated property. NASA's definition of life¹ focuses on the type of substrate and the ability to evolve, but equally striking is the ability of living organisms to thrive independently. Even simple bacteria have immune systems [2] and self-repair mechanisms [5] to fend off infection and recover from injuries. They autonomously control their behavior such as searching for nutrients and initiating reproduction. Additionally, they communicate among themselves to coordinate collective behavior [11], all while facing challenging, unstructured environments. In contrast, human-made "autonomous" systems often require intervention when facing new challenges [13].

Cells, in general, are underappreciated when compared to extant macro-organisms and their complex features, such as the human brain. While the human brain is frequently used as an example of biological complexity, a single cell—the zygote—encodes the developmental rules that produce the brain as well as the rest of the body. Moreover, it is individual cells, neurons, which are the real agents behind the brain's network plasticity and learning capacity, deciding which connections to form and strengthen and selecting memories to store intracellularly.

What governs individual cells and enables their autonomy—whether they are bacteria, zygotes, or neurons in the brain—is their sensory-regulatory network. This self-assembling web of electrochemical interactions among numerous chemical compounds forms what is essentially a flexible and robust control system [8, 15]. Such self-tuning, decentralized, and adaptive natural controllers are not only a marvel of evolution but also the dream of every control engineer.

Regulatory networks were likely integral to the origin of life approximately 4 billion years ago, and evolution has been refining their designs ever since. If we trace the human lineage in terms of number of generations,² then for 99.66% of our evolutionary history, we were single-celled organisms—initially prokaryotes, then eukaryotes. For an additional 0.32% of our evolution, we were animals without a central nervous system, i.e., without a brain. The first brain appeared only in the last 0.02% of our evolutionary timeline, likely as a simple ganglion in a worm-like organism the size of a grain of rice [4]. The human brain diverged from the chimpanzee's only in the last 0.0000007% of our evolution.

Over the long term, change in design due to evolution is proportional to the length of the evolutionary period [14]. It is therefore clear that the great majority of our evolutionary design occurred at the single-cell level. Once our single-cell ancestors acquired the ability to self-organize into multicellular bodies, the brain readily evolved, requiring relatively few cellular design changes. To use a genetic

¹ "A self-sustaining chemical system capable of Darwinian evolution."

² This is a better measure than time due to the large differences in generation times among the organisms in our ancestry, from hours (bacteria) to decades (hominins). Since genetic variation occurs between generations, tracking evolution in terms of generations provides a more accurate reflection of evolutionary changes.

programming analogy, the program for our last single-cell ancestor contained most of the design necessary to build a human brain.

For a discussion on the choice of the term design, see Box 17.1.

Genetic programming was inspired by biological evolution and genetics, and it has excelled at solving a wide range of problems, from scientific modeling to evolving programs for quantum computers and robot control systems. Nevertheless, it is generally agreed that we are far from reaching the full potential of the field.

Genetic programming suffers from limitations that nature does not—or, at least, not to the same extent. For instance, we frequently encounter obstacles that prevent the evolution of solutions beyond a certain level of complexity [9]. This can stem from prohibitive computational burdens, a lack of generalization, and limitations in the genetic representation that hinder modularity and evolvability, all of which impact scalability [9].

One solution that is often suggested is that we emulate biology more closely [7, 9]. A key feature of life that enables its extraordinary evolvability and scalability is the natural genetic representation. This representation, which evolved alongside cells, encodes control systems—namely, the cells’ regulatory networks—that possess flexible topologies with significant modularity and redundancy, allowing them to achieve remarkable levels of complexity and adaptability. To create similarly complex and autonomous control systems, we must focus on the same mechanisms that underlie these properties in nature—specifically, cell regulatory networks.

17.1 Evolution of Design

Daniel Dennett referred to Darwinian evolution as design without a designer [3]. When prokaryotic cells evolved for billions of years before giving rise to eukaryotic cells, they were evolving their design, which when the fostering circumstances arose allowed eukaryogenesis. Similarly, when eukaryotic cells evolved for a billion years before giving rise to metazoans, they were evolving the design that eventually made metazoan multicellularity possible.

We use the term design, instead of information or complexity, because both genetic information and organismal complexity can increase without making an organism’s design any more adaptive, while it is possible to make the design more adaptive by simplifying it and reducing the amount of encoding information. The longest genome on record belongs to an unremarkable fern, topping 160 Gb, or 50X the human genome.

In the genetic algorithms and AI communities, this concept could correspond to the term “learning.” However, using the word learning this way can create confusion, since it already has a clear meaning in biology and cognitive science, that conflicts with this liberal use.

“Design (noun) 5a: “an underlying scheme that governs functioning, developing, or unfolding”;” 2024, <https://www.merriam-webster.com/dictionary/design>.

Elfa, the evolutionary platform used in this work, aims to narrow the gap between genetic programming and biology [10]. Elfa was developed with two primary goals: to serve as a high-fidelity evolutionary platform for studying the evolution of cell regulatory networks and understanding cell behavior, and to explore how this biologically inspired model can be used as a framework for evolving autonomous control systems for industrial applications.

Elfa's genetic representation is closely modeled after the natural genetic representation of prokaryotic cells. What primarily differentiates Elfa from other evolutionary platforms based on gene regulatory networks is its level of detail and integration of mechanistic interactions and stochasticity. This level of granularity strikes a balance between capturing the essential features of cell regulatory systems and maintaining computational tractability.

For the present work, we chose a biologically plausible, multi-objective optimization problem: evolving populations of primitive cyanobacteria in a variable light environment in an open-ended manner. In this scenario, we model a population of cells where the expression levels of genes can evolve and affect various traits, such as cellular target size, growth rate, division rate, photosynthetic activity, and somatic mutation rate. Additionally, proteins can evolve to sense environmental cues and dynamically regulate gene expression and protein activity, allowing evolution to explore complex regulatory regimes like feedback loops.

This is a challenging problem because it requires optimizing traits that are highly epistatic in response to natural processes that are noisy and often non-linear. This biological scenario allows us to test the Elfa platform against known biological behavior, making any large discrepancies with natural organisms readily apparent. Additionally, its similarity to many industrial process control problems highlights the potential for applying our methodology to solve those problems as well.

At the start of our experiments, we seeded the populations with cells that expressed their genes at a low, constant rate and had no regulatory ability. We exposed these populations to an environment where light irradiance followed a daily cycle. Each cell also contained chemical signals whose concentration correlated with the light intensity in the environment, as well as the cell's level of stress and energy reserves.

Our hypotheses were:

1. Over the course of evolution, the cell populations would evolve faster replication rates and optimize their photosynthetic activity in response to the available light.
2. The Elfa platform is sufficient to capture evolutionary behavior in silico.

Populations could reach higher fitness via two paths. The first was due to mutations that adjusted the fixed rate at which genes were expressed, without incorporating any dynamic regulation mechanisms. The second was the evolution of regulatory mechanisms that allowed the dynamic regulation of gene expression, growth, and cell division in response to environmental and stress conditions.

Our results support our hypothesis. All populations evolved higher fitness over time by adjusting gene expression levels. Additionally, a small percentage of the populations, ranging from 2 to 19% depending on the environment, evolved dynamic regulation of genes or proteins. There was a wide variety of regulatory solutions, but

most regulated cells adapted to the light cycle by concentrating their growth and reproduction during the day and reducing protein expression at night. This behavior allowed them to harvest substantially more energy from the environment and achieve much higher fitness than non-regulated cells, replicating up to four times faster.

Finally, we found a surprising positive correlation between the environment's latitude and the rate at which regulation fixed in populations. Populations evolving at 60° latitude, where seasonal light variations are more pronounced, fixed regulation ten times as often as those evolving at the Equator.

17.2 Methods

17.2.1 *Elfa Digital Evolution Platform*

Elfa, the digital evolution platform used in this study, was inspired by biology and developed by carefully translating the genetic representation of prokaryotic cells into a digital format. Elfa was designed to capture the essential features of the cell's chemical-based sensing and regulatory mechanisms, including much of the inherent stochasticity, while taking advantage of the digital format to optimize other processes, thus avoiding the steep costs of chemical simulations.

Elfa is an agent-based system, where each agent is a digital model of a prokaryotic cell. Each cell contains a single circular chromosome, represented by a binary string. The circular nature of the chromosome allows any genetic feature encoded in the string to loop back to the beginning after reaching the end. The length of the chromosome is not fixed and can change during evolution due to insertion and deletion mutations.

Elfa's genetic representation encodes a typical set of genetic and regulatory elements from molecular biology, including basal promoters, regulatory binding sites, transcription start sites, and gene termination sites. Genes encode different types of proteins that perform a variety of functions, such as RNA polymerase, responsible for transcribing genes; transcription factors, which bind to regulatory sites on the chromosome and increase or decrease gene transcription; and proteases, which recycle other proteins.

In Elfa, cells have a prototypical bacillus shape—cylindrical with hemispherical caps at both ends—and contain a given volume of cytoplasm bounded by a semi-permeable membrane. The cell's volume and surface area change as it grows and divides. Both growth rate and target size are evolutionarily controlled through the expression of growth and division factors. The cell's virtual cytoplasm contains the single circular chromosome, proteins, ligands (small molecules that bind to proteins and convey signals), and other molecules, all of which can interact.

Elfa uses ligands to provide each cell with signals about the external environment and the cell's internal state. The strength of these signals is proportional to the concentration of the corresponding ligand. Proteins can evolve a form of regu-

lation called allostery by acquiring ligand-sensitive surfaces that temporarily bind ligands based on their affinity sequence, influencing the protein's function by either increasing or decreasing its activity.

The cell's binary chromosome is organized into 32-bit segments, each corresponding to an integer number. The combination of 32-bit segments within a certain range of values encodes operons, each containing one or more genes. Operons can include one or more basal promoters, and have one or more upstream regulatory binding sites. Segments with integer values outside of the coding range are interpreted as either linker domains or non-coding sequences, depending on whether or not they are transcribed.

During replication, the cell's chromosome can suffer mutations. These can be simple point substitutions (bit flips), as well as multiple segment duplications, transpositions, and deletions. Consequently, the length of the chromosome, the number of genes, and their functions, all can evolve. For instance, mutations in an operon's basal promoter can alter its affinity for RNA polymerase, thereby changing the expression level of that operon's genes. Mutations can also change the properties of proteins encoded by genes, for example, making them sensitive to a ligand. In addition, mutations can cause genes to duplicate or lose function. In Elfa, much like in biology, mutations can reshape the architecture of chromosomes, which encode not only genes but also the regulatory regions that determine how those genes are expressed.

Proteins have a half-life, determining the average time they remain in the cell before being recycled. Cells, however, have no age limit but accumulate harmful protein aggregates (plaque) over time [1].

Plaque forms when proteins are damaged due to harmful metabolic byproducts (see ROS in Table 17.2) and cannot be recycled, resulting in their accumulation in aggregates. These aggregates can, in turn, cause further damage to healthy proteins, leading to cell "aging." When a cell replicates, all of its plaque is segregated into one of the daughter cells, while the other is born "rejuvenated" [12]. However, at cell division, all proteins and energy reserves of the mother cell are divided between the two daughter cells.

There are several costs and constraints to the cell's actions. Both cell growth and chromosome replication incur energy and time costs, proportional to the volume and length added, respectively. If a cell runs out of energy, it dies.

Transcribing genes and translating proteins also incur energy costs and time delays, proportional to the gene's length and the protein copy number. This dynamic creates selective pressure for efficient genetic encoding and regulation. By imposing realistic costs and constraints on the cell, while providing sensory feedback signals, we avoid introducing artificial limitations, such as maximum cell age or genome size, and configuration parameters, such as somatic mutation rate or transcription rate. Leaving more variables under evolution allows for a greater variety and complexity of regulatory solutions.

17.2.2 Seeding Population

The ancestral genotype that seeded all experimental populations possessed only four genes, each transcribed at constant, but different, rates proportional to their basal promoter affinity (Fig. 17.1). These genes encoded a photosynthetic complex, an RNA polymerase, a growth factor, and a division factor, described in Table 17.1. In addition to these four gene products, Elfa’s representation allows the encoding of proteins with the function of transcription factors. Transcription factors bind to enhancers and either promote or repress protein transcription of the operon downstream.

Populations are well mixed and capped in size. Once a population reaches the maximum size, the birth of any additional cell results in the random removal of another cell. This spatial constraint creates a strong selective pressure for fast replication. When a cell divides, both daughter cells inherit any somatic mutations the mother cell may have acquired. Additionally, each daughter cell independently faces the risk of new mutations arising from the replication process.

17.2.3 Ligands in Elfa

For this study, Elfa was configured to supply each cell with five different ligands, each conveying a different signal. The choice of number and type of ligands was meant to provide a redundant but not too numerous set of signals that were biologically plausible, and informative, in case regulation evolved. However, the ancestral genotype was insensitive to these signals. To sense them, cells had to first evolve proteins with ligand-sensitive domains with affinity for the specific ligands. The five types of ligands used in this experiment are described in Table 17.2.

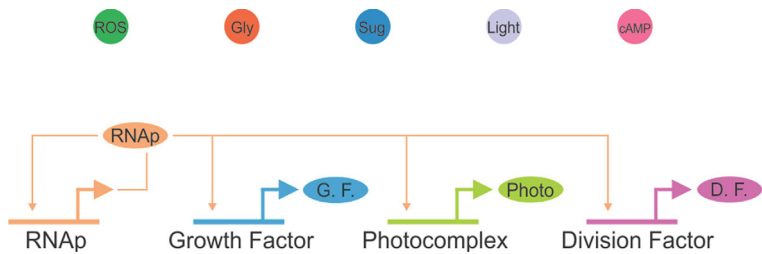


Fig. 17.1 Non-regulated seeding cell. In this cell, the four genes are expressed at different, but constant, rates induced only by the RNAp protein (orange regulatory arrows). None of the genes are sensitive to any of the five ligands (circles), indicated by the lack of arrows

Table 17.1 List of the genes included in the ancestral genotype that seeded all experiments

Product	Description
Photosynthetic complex	This complex is responsible for converting CO ₂ , H ₂ O, and light into sugar, which the cell uses for energy. The sugar conversion rate in a cell is proportional to the abundance of the photosynthetic complex and the light irradiance at a given time, and it is limited by the amount of CO ₂ that diffuses into the cell. If the cell runs out of CO ₂ in the presence of light, this complex produces a harmful compound—Reactive Oxygen Species (ROS)—instead of sugar. At high concentrations, ROS slows protein translation and causes damage to the cell’s components, including somatic mutations
RNA polymerase (RNAP)	This protein is responsible for transcribing the cell’s genes. The rate of transcription for a gene depends on the abundance of RNAP and its affinity for the gene’s promoter(s)
Growth factor	This factor regulates cell growth. When the concentration of this factor surpasses its previous maximum, the cell adds a volume proportional to the difference between the current concentration and the previous maximum. There is no limit to how large a cell can grow before dividing. As a result, different cell target sizes can evolve by independently evolving growth and division rates
Division factor	This factor is responsible for both chromosome replication and initiating cell division. When the abundance of this factor surpasses its previous maximum, the replicating chromosome adds a length proportional to the increase in abundance. Once the abundance reaches a set threshold, the cell divides into two daughter cells (Fig. 17.2)

17.2.4 Experimental Parameters

For these experiments, we evolved between 120 and 160 population replicates per condition, with each population capped at 200 cells. Evolution lasted for 365 virtual days, after which populations were incubated for an additional 20 generations with mutations turned off, except for somatic mutations which cells could still suffer due to their own metabolism. Cells experienced a light environment equivalent to a depth of 4.0 m of ocean water at the experiment’s specified latitude, which varied between 0, 40, and 60°. Half of the populations started with light levels corresponding to 6:00

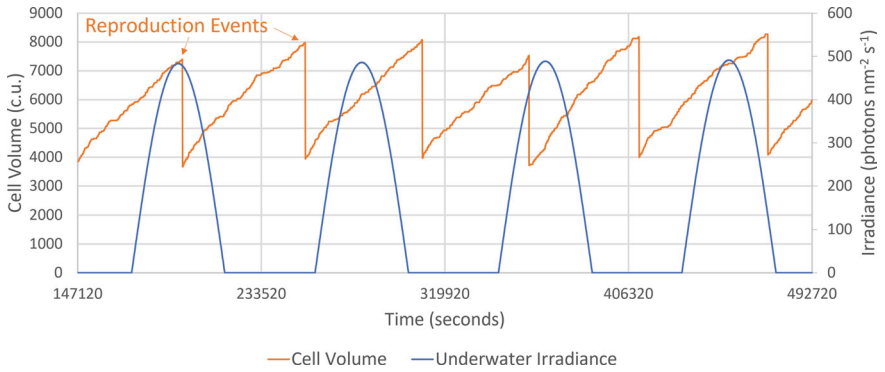


Fig. 17.2 Behavior of the non-regulated seeding cell. This plot compares the growth of the cell measure by volume (left y-axis, brown) with the light irradiance level (right y-axis, blue) over time. Sharp drops in cell volume indicate cell division events, which occurred 1.5 times per 24 h period, on average, with an expected birth volume of 4000 units

Table 17.2 List of signaling ligands available to cells during the experiments

Ligand	Description
Reactive Oxygen Species (ROS)	This ligand signals the amount of this harmful compound. It is produced as the cell runs out of CO ₂ during photosynthesis. Higher concentrations of ROS increase the probability of somatic mutations, protein damage, translation slowdown, cell aging, and loss of glycogen reserves
Cyclic AMP (cAMP)	This ligand signals starvation stress. It is produced as the cell runs out of sugar reserves. At low concentrations, it indicates that glycogen reserves are being converted into sugar. However, in high concentrations, it indicates that the glycogen reserves are running out. A high concentration of cAMP increases the chance of somatic mutations, protein damage, and translation slowdown
Irradiance level	This ligand indicates the amount of light reaching the cell. Its concentration follows a 24-h cycle of night and day, where underwater daylight irradiance is modeled according to the literature [6]
Sugar reserves	This ligand indicates the amount of sugar immediately available as energy for the cell. It is produced as the cell performs photosynthesis and accumulates sugar
Glycogen reserves	This ligand indicates the amount of energy in long-term storage. It is produced when the sugar reserves reach a maximum threshold, and any excess sugar is converted into glycogen

Table 17.3 Experiments’ mutation rates—a chromosome segment is a 32-bit binary string

Mutation type	Mutation probability
Point mutation	1/50 per segment per generation
Segment duplication	1/1,200 per segment per generation
Segment deletion	1/1,200 per segment per generation
Segment transposition	1/3,000 per segment per generation

a.m. on January 1, while the other half started at 6:00 a.m. on July 1. Mutation rates were set according to Table 17.3.

17.2.5 Analyses

At the end of the experiment, we sorted the populations based on whether they had evolved and fixed allosteric regulation, transcription factors, or neither. Elfa provides population reports that make this classification straightforward. In addition, we collected samples from each of the final populations that evolved allosteric regulation and mapped their regulatory networks.

To measure the fitness of a given population, we collected sample cells and incubated them for an additional 30 simulated days with mutations turned off (except for somatic mutations) to assess their viability. Due to the likelihood of somatic mutations under stress conditions, populations typically contained a number of damaged and unviable cells. This issue was more pronounced in populations that did not evolve regulation, as these cells were insensitive to stress signals. As a result, measuring the fitness of a given population was a computationally intensive process.

Once we had identified the fitness of a set of populations and analyzed the regulatory networks of their representative genotypes, we selected a few cells that exemplified the different behaviors that evolved. We then plotted their daily cycles of growth and division, noting any salient dynamics. These plots were based on reports that Elfa can generate for individual cells or lineages.

17.3 Results

17.3.1 Experiment 1

In our first experiment, we evolved a set of 160 populations at 0° latitude for a period of 365 d. At the end of the experiment, all populations had evolved faster replication rates than the seeding ancestor, but only three populations fixed allosteric regulation. The regulated populations, however, had substantially faster replication rates than the non-regulated ones (Figs. 17.3 and 17.4).

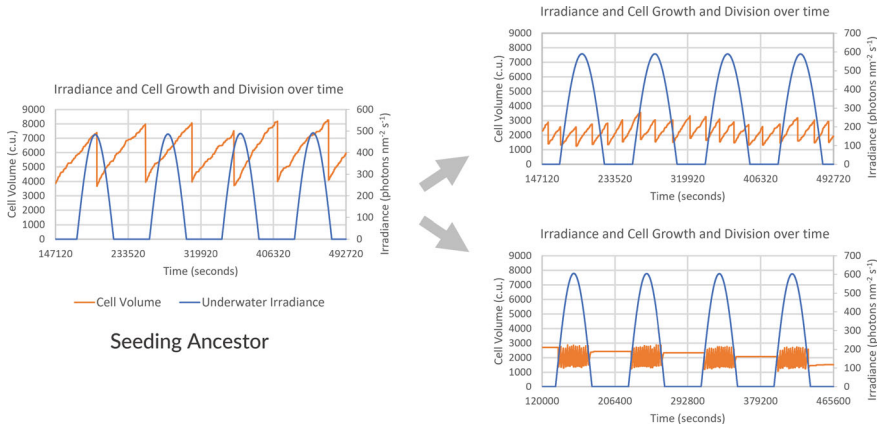


Fig. 17.3 Experiment 1: Evolution at 0° latitude. **(Left):** The growth behavior of a typical unregulated ancestral cell, visualized as changes in cell volume over time, overlaid with light irradiance. **(Right):** The growth behavior of two example cells from evolved populations, both of which replicate faster and maintain a smaller average cell size than the ancestor. Notably, the example from the population that evolved allosteric regulation (**bottom**) exhibits more rapid replication, constrained to the daylight period, compared to the cell from the non-regulated, evolved population (**top**)

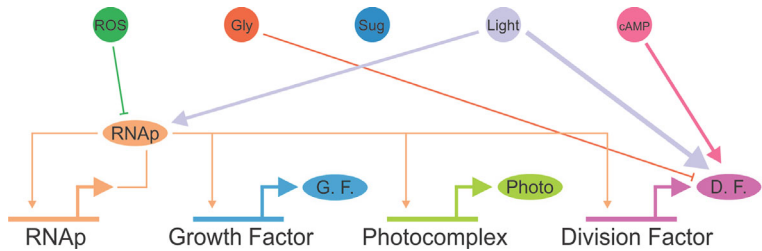


Fig. 17.4 Gene regulatory network of a cell from a population that evolved at 0° latitude. Compared to the ancestral cell (Fig. 17.1), this cell relies primarily on the light intensity, conveyed by the Irradiance ligand, to increase the activity of the RNAp and the division factor (purple arrows). Additionally, starvation, conveyed by the cAMP ligand, also enhances cell division (pink arrow), while a high glycogen level represses it (dark orange arrow). Finally, oxidative stress, conveyed by the ROS ligand, represses the activity of the RNAp (green arrow). The resulting behavior is depicted at the bottom right of Fig. 17.3 and equates to a replication rate of 14.5 divisions per 24 h

17.3.2 Experiment 2

We selected a representative genotype from one of the populations that evolved regulation in Experiment 1 and incubated it for 30d at different latitudes (ranging from 20° to 70°, in 10° intervals) and different times of the year. In all cases, the cells survived and divided normally, though they exhibited higher cAMP levels with increasing latitudes, indicating starvation stress (Figs. 17.5 and 17.6).

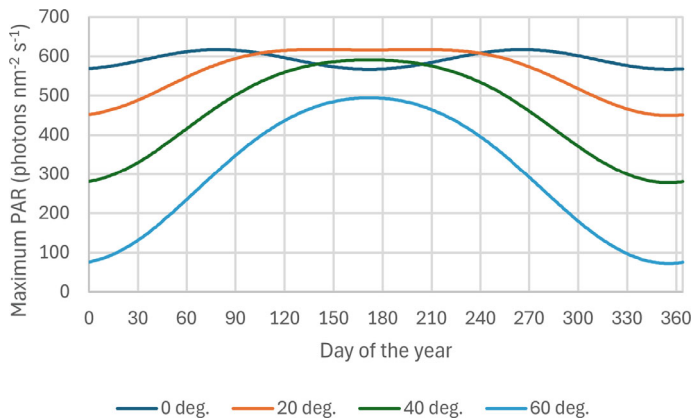


Fig. 17.5 Maximum daily Photosynthetically Active Radiation (PAR) throughout the year at different latitudes. Note that at the Equator, the seasonal variation in light irradiation is minimal, but it becomes more pronounced at higher latitudes

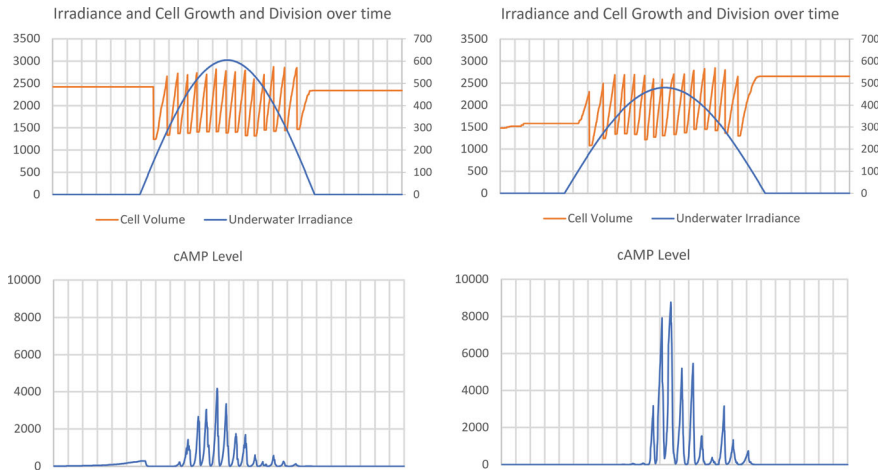


Fig. 17.6 Incubation of a regulated genotype that originally evolved at 0° latitude. **(Left)** The cells' behavior at 0° latitude plotted against irradiance **(top)** and starvation stress level, as indicated by cAMP **(bottom)**. The x-axis shows time over a 24-h period. **(Right)** Cell's behavior and starvation stress level at 50° latitude. Notably, at 50° latitude, there is higher starvation stress, conveyed by the cAMP ligand levels, but in either case the cells grow and divide normally

Remarkably, the regulatory strategy that evolved at 0° latitude—where seasonal differences in daylight are minimal—generalized effectively to higher latitudes with more pronounced seasonal variations, despite these populations never having been exposed to such conditions during evolution. When we repeated this incubation using representative genotypes from non-regulated populations, the cells died within a few generations.

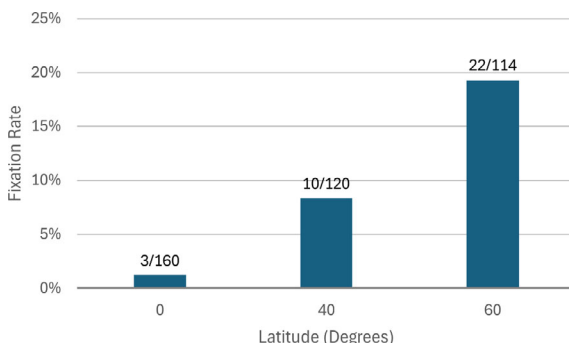


Fig. 17.7 Comparison of the rates of regulation fixation at different latitudes. The fixation rate (y-axis) is calculated by dividing the total number of populations where regulation by allostery fixed over the total number of surviving populations in the experiment. There is a significant trend toward higher rates of fixation of regulation with increasing latitudes ($\chi^2 = 25.031$, $p = 0.00000367$)

17.3.3 Experiment 3

We repeated Experiment 1 at higher latitudes, 40° and 60°, evolving 120 populations in each, to test whether evolution under higher seasonal variation leads to different regulatory solutions (Fig. 17.5). What we found was that the number of populations that fixed allosteric regulation increased significantly at both 40° and 60° when compared with 0° latitude. In fact, there is a significant positive correlation between fixation rates and latitude ($\chi^2(2, N = 394) = 25.031$, $p = 0.00000367$) (Fig. 17.7).

Notably, at 60°, 6 out of 120 populations went extinct, all during the summer. This was due to a dynamic observed exclusively at this latitude, where cells replicated rapidly during the long summer days but often failed to survive when irradiance dropped at the end of the day. While most populations survived this bottleneck, a few did not.

In addition, we observed a great variety of regulatory solutions. For example, some populations used positive regulation, that is, they relied on environmental signals to increase their activity (Fig. 17.8). Other populations evolved the opposite, negative regulation (Fig. 17.9). One population relied solely on one ligand (Fig. 17.10), while the remaining regulated populations used two or more ligands.

One population evolved a strategy where cells would grow very large at night, without dividing, and at day time they would divide several times in sequence, maintaining a smaller size (Fig. 17.11). This was reminiscent of the behavior of the green alga *Chlamydomonas*, which grows during the day without dividing, and at night divides one or more times in succession.

Most of the other strategies we observed, however, relied on slow or no growth at night, and rapid growth and division during the day.

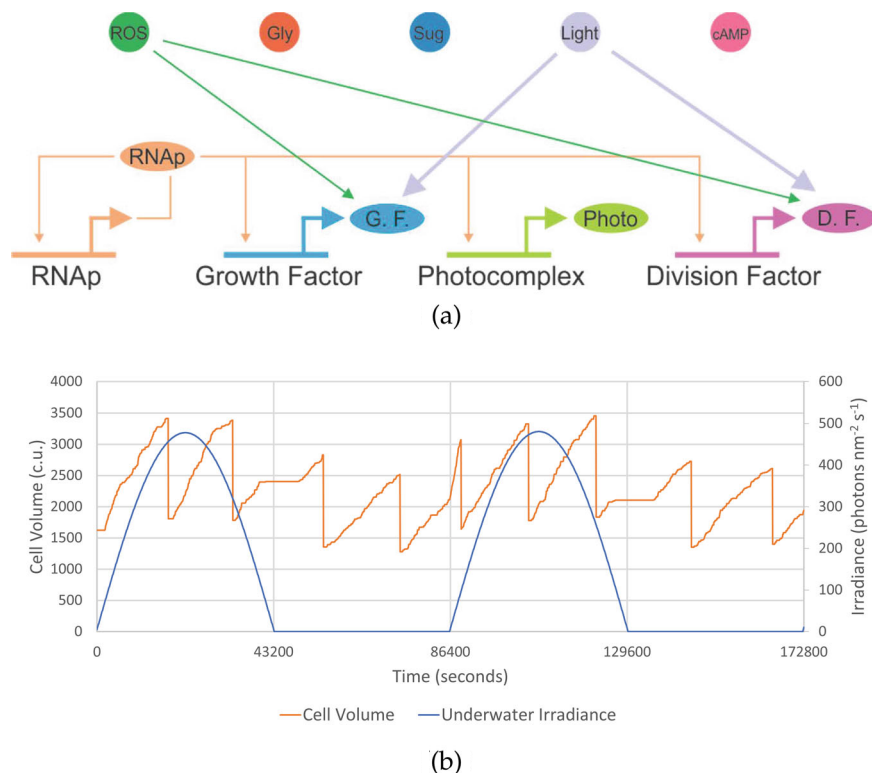


Fig. 17.8 Example of a cell with only positive regulation. **a** The regulatory network of a cell, taken from a population which evolved at 40° latitude, in which oxidative stress (ROS, green) and irradiance (Light, purple) both positively regulate (arrows with pointed heads) growth and division. **b** The growth of the cell measured by volume (left y-axis, brown) with the light irradiance level (right y-axis, blue) over a 2-day period. This cell grows faster, and larger during daylight. It replicates on average 4.5 times every 24 h

17.4 Discussion

Environmental variation is well known to foster the evolution of flexible phenotypes, so it was not surprising that our experimental results support the hypothesis that, under variable light conditions, cell populations would evolve to use environmental signals to optimize replication and photosynthetic activity. However, this unsurprising result supports our second hypothesis that Elfa, our new and fine-grained evolutionary platform, is capable of capturing evolutionary behavior in a challenging multi-optimization problem. Just to mention a few of the challenges:

Protein production and cell replication incurred costs of both energy and time. Gene expression rates could be adjusted if regulation evolved, but the optimal level of expression at any given time depended on noisy, non-linear dynamics. Evolutionary

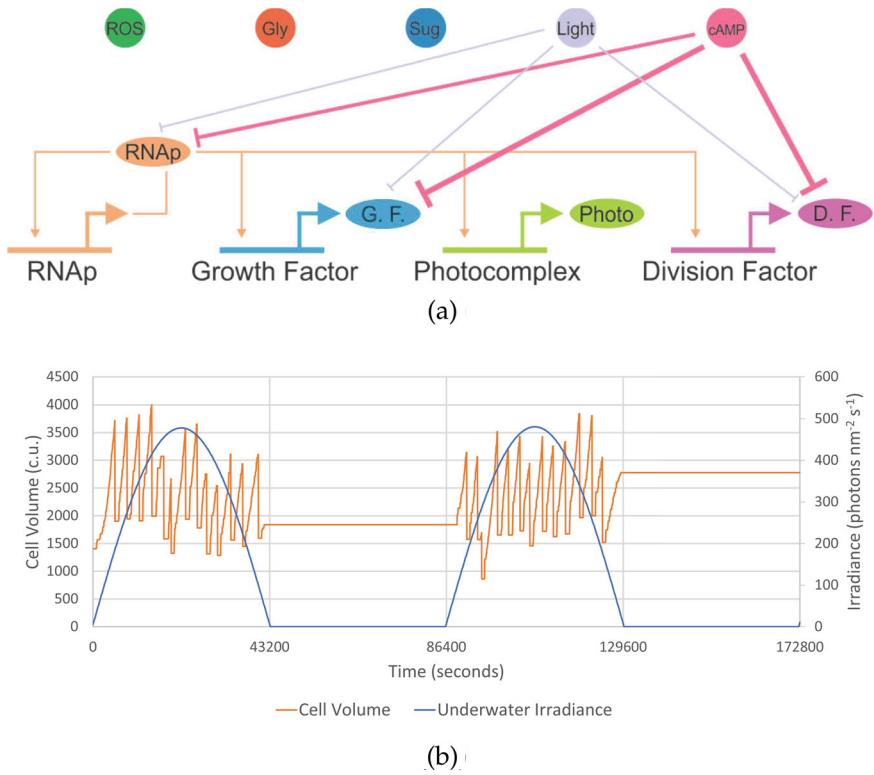


Fig. 17.9 Example of a cell with only negative regulation. **a** The regulatory network of a cell, taken from a population which evolved at 40° latitude, in which irradiance (Light, purple) and oxidative stress (CAMP, pink) negatively regulate (arrows with flatheads) growth and all major cell function except for the photosynthetic complex. **b** The growth of the cell measured by volume (left y-axis, brown) with the light irradiance level (right y-axis, blue) over a 2-day period. This cell only grows and divides during daylight, and it does not display a consistent target volume. It replicates on average 13.5 times every 24 h

adaptation often required compensatory mutations, as altering the expression rate of any gene had epistatic effects. Cell target volume could evolve by combining changes in growth and cell division rates, but cell volume also affected global transcription rate, with smaller cells transcribing more slowly than larger cells given the same chromosome size and RNA polymerase concentration, due to molecular crowding effects. As in biology, changing a parameter in our model often affected other ones.

In addition, the fact that regulation evolved and fixed within a few hundred generations, despite the many interacting parameters, supports the argument that more closely emulating the natural genetic representation in evolutionary computation systems could improve evolvability.

One notable result was the significant correlation between the fixation of regulation and latitude (Fig. 17.7). It is plausible that at 40° latitude, there is stronger

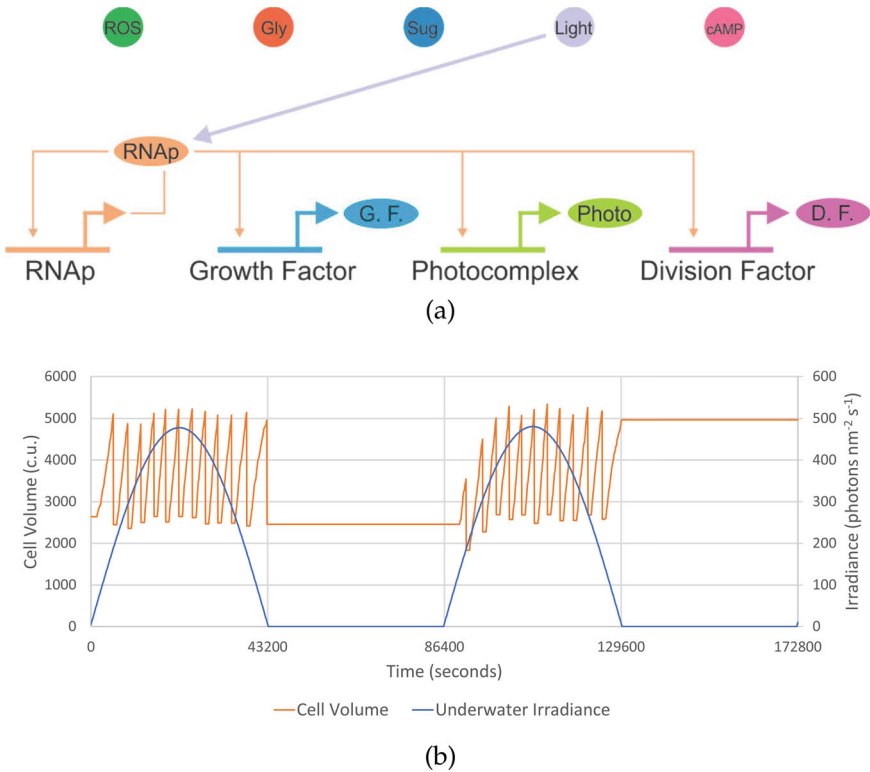


Fig. 17.10 Example of a cell with a regulatory network consisting of a single regulatory interaction. **a** The regulatory network of a cell, taken from a population which evolved at 40° latitude, in which irradiance (Light, purple) positively regulates only RNA polymerase, which in turn regulates the remaining cell functions. **b** The growth of the cell measured by volume (left y-axis, brown) with the light irradiance level (right y-axis, blue) over a 2-day period. Notably, although this cell exhibits only a single new regulatory interaction, because it targets a master regulator (RNAp), the one interaction has an outsized impact on overall cell behavior, similar to a more complicated solution (see Fig. 17.9). This demonstrates the importance of “hub” regulators in networks. This cell displays a fairly consistent target volume and it replicates on average 12.6 times every 24 h

selective pressure for adaptive regulated phenotypes than at the Equator, due to the combined effects of daily and seasonal variations in light irradiance. However, this trend continued toward higher latitudes. At 60°, the difference between summer and winter is so extreme, and the environment so inhospitable, that 6 out of 120 populations went extinct (Fig. 17.5). Yet, regulation fixed at 60° at a rate roughly twice as high as at 40°. This intriguing result warrants further investigation to understand which aspects of environmental variation are intensifying selective pressure. This is particularly important given that many natural processes are responses to environmental variation, and evolutionary experiments are often designed around variable conditions.

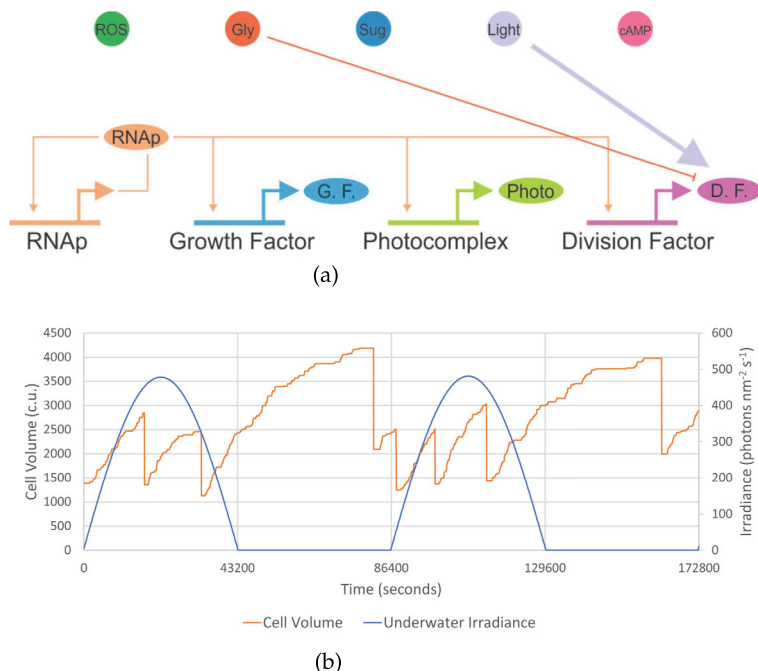


Fig. 17.11 Example of a cell that isolates division to the day period. **a** The regulatory network of a cell, taken from a population that evolved at 40° latitude, in which high glycogen level (dark orange) represses cell division while irradiance (Light, purple) strongly activates (pointed arrow) division. **b** The growth of the cell measured by volume (left y-axis, brown) with the light irradiance level (right y-axis, blue) over a 2-day period. Notably, because of the strong induction of division by light, division events only occur during the day period, leading to the expansion of cell volume to nearly twice the normal level during the night. This cell replicates on average 4.3 times every 24 h

We observed a great diversity in the regulatory solutions that evolved. Even ostensibly equivalent behaviors were often based on very different regulatory networks (Figs. 17.9 and 17.10). This highlights the role of historical contingency in evolution, suggesting that many important regulatory circuits found in nature may exist in their current form simply because they were the first to become fixed during evolution.

When looking at the different regulatory networks that evolved, it is important to keep in mind that the simplicity of the diagrams can be misleading. Each line in the diagram can represent a range of complex interactions. For instance, when we say that species A activates species B, we leave out important details such as the concentrations of each species, the strength and duration of their interaction, and the dynamic nature of these variables, such as the half-life of the molecules involved. Additionally, when two or more species act on the same target, their effects do not simply add or subtract—they also compete, and both are subject to diminishing returns and short-term stochastic fluctuations, resulting in intermittent activity that

can differ significantly from the long-term average. Therefore, the simplicity of our diagrams hides a wealth of information that is difficult to represent but is fundamental to understanding the complexity of controlling a cell's fate in a variable environment.

Finally, we found that transcription factor regulation was absent from the set of regulatory solutions that became fixed. While this form of regulation frequently evolved in most populations, it never reached the point of fixation. This is intriguing, as many natural gene regulatory circuits rely on transcription factor regulation. In our specific problem, allosteric protein regulation may have been sufficiently effective and energetically cheaper, as it eliminates the need to produce additional transcription factor proteins.

17.4.1 Future Experiments

We plan to build on the current environmental model by incorporating additional sources of variation, such as cloud coverage and self-shading, where cells higher in the water column partially block light for those below. We also intend to give cells more degrees of freedom, including the ability to move vertically within the water column, and scavenge nutrients from dead cells, including DNA fragments, which could also be used for horizontal gene transfer (HGT).

We are interested in HGT as a mechanism to increase evolvability, but its potential to shape chromosome evolution is also intriguing. Along these lines, we plan to leverage Elfa's fine-grained genetic architecture and perform experiments on whole genome duplications and study how such events affect cells in the short term and over evolutionary time.

17.4.2 Potential Applications in AI and Autonomous Systems

As mentioned earlier, we consider Elfa a dual-purpose platform. The same model that allows the evolution of virtual cells' regulatory systems can be applied to evolving autonomous control systems for industrial applications. For example, living cells routinely produce different chemicals under tight regulation, a capability that could be replicated in virtual cells to control industrial chemical reactors. These reactors are typically equipped with human-designed control systems, but evolution is known for producing novel solutions that rival state-of-the-art designs. Since Elfa can already evolve regulatory systems with complexity comparable to many industrial control systems, it would be interesting to compare their performance and practicality.

Another line of research we are interested in is evolving cells that self-assemble into networks and solve problems collectively. This would be valuable for studying the evolution of multicellularity, but it could also lead to better artificial neural networks where connections remain plastic throughout life, and where nodes can store memory internally, much like natural neuronal networks.

Acknowledgements Thanks to Stephan Winkler, Ting Hu, Alex Lalejini, Wolfgang Banzhaf, Christopher Cray, Bogdan Burlacu, and all the participants of Genetic Programming Theory and Practice XXI.

References

1. Bednarska, N.G., Schymkowitz, J., Rousseau, F., Van Eldere, J.: Protein aggregation in bacteria: the thin boundary between functionality and toxicity. *Microbiology* **159**(9), 1795–1806 (2013). <https://doi.org/10.1099/mic.0.069575-0>. <http://mic.microbiologyresearch.org/content/journal/micro/10.1099/mic.0.069575-0>
2. Bernheim, A., Sorek, R.: The pan-immune system of bacteria: antiviral defence as a community resource. *Nat. Rev. Microbiol.* **18**(2), 113–119 (2020). <https://doi.org/10.1038/s41579-019-0278-2>. <https://www.nature.com/articles/s41579-019-0278-2>. Publisher: Nature Publishing Group
3. Dennett, D.C.: *Darwin’s Dangerous Idea: Evolution and the Meanings of Life*. A Touchstone Book. Touchstone, New York (1996). OCLC: 258630142
4. Evans, S.D., Hughes, I.V., Gehling, J.G., Droser, M.L.: Discovery of the oldest bilaterian from the Ediacaran of South Australia. *Proc. Natl. Acad. Sci.* (2020). <https://doi.org/10.1073/pnas.2001045117>. <http://www.pnas.org/content/early/2020/03/17/2001045117>. Publisher: National Academy of Sciences Section: Biological Sciences
5. Hu, Z., Bai, X.: Self-repair and resuscitation of viable injured bacteria in chlorinated drinking water: *Achromobacter* as an example. *Water Res.* **245**, 120,585 (2023). <https://doi.org/10.1016/j.watres.2023.120585>
6. Kirk, J.T.O.: *Light and Photosynthesis in Aquatic Ecosystems*. Cambridge University Press (2010)
7. Miikkulainen, R., Forrest, S.: A biological perspective on evolutionary computation. *Nat. Mach. Intell.* **3**(1), 9–15 (2021). <https://doi.org/10.1038/s42256-020-00278-8>. <https://www.nature.com/articles/s42256-020-00278-8>. Number: 1 Publisher: Nature Publishing Group
8. Muzzey, D., Gómez-Urbe, C.A., Mettetal, J.T., van Oudenaarden, A.: A systems-level analysis of perfect adaptation in yeast osmoregulation. *Cell* **138**(1), 160–171 (2009). <https://doi.org/10.1016/j.cell.2009.04.047>. <https://www.sciencedirect.com/science/article/pii/S009286740900508X>
9. O’Neill, M., Vanneschi, L., Gustafson, S., Banzhaf, W.: Open issues in genetic programming. *Genet. Program. Evolvable Mach.* **11**(3–4), 339–363 (2010). <https://doi.org/10.1007/s10710-010-9113-2>
10. Pontes, A.C.: *The Evolutionary Origins of Cognition: Understanding the Early Evolution of Biological Control Systems and General Intelligence*. Michigan State University (2021)
11. Prindle, A., Liu, J., Asally, M., Ly, S., Garcia-Ojalvo, J., Süel, G.M.: Ion channels enable electrical communication in bacterial communities. *Nature* **527**(7576), 59–63 (2015). <https://doi.org/10.1038/nature15709>. <http://www.nature.com/articles/nature15709>. Number: 7576 Publisher: Nature Publishing Group
12. Stewart, E.J., Madden, R., Paul, G., Taddei, F.: Aging and death in an organism that reproduces by morphologically symmetric division. *PLOS Biol.* **3**(2), e45 (2005). <https://doi.org/10.1371/journal.pbio.0050045>

- 1371/journal.pbio.0030045. <http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.0030045>
13. Tener, F., Lanir, J.: Investigating intervention road scenarios for teleoperation of autonomous vehicles. *Multimed. Tools Appl.* **83**(21), 61103–61119 (2024). <https://doi.org/10.1007/s11042-023-17851-z>. <https://doi.org/10.1007/s11042-023-17851-z>
 14. Worden, R.P.: A speed limit for evolution. *J. Theor. Biol.* **176**(1), 137–152 (1995). <https://doi.org/10.1006/jtbi.1995.0183>. <https://www.sciencedirect.com/science/article/pii/S002251938570183X>
 15. Yi, T.M., Huang, Y., Simon, M.I., Doyle, J.: Robust perfect adaptation in bacterial chemotaxis through integral feedback control. *Proc. Natl. Acad. Sci.* **97**(9), 4649–4653 (2000). <https://doi.org/10.1073/pnas.97.9.4649>. <http://www.pnas.org/content/97/9/4649>

Chapter 18

How to Measure Explainability and Interpretability of Machine Learning Results



Elisabeth Mayrhuber, Bogdan Burlacu, and Stephan M. Winkler

Abstract The increasing complexity of machine learning models has motivated the need to ensure that the results are understandable and transparent, enabling trust and accountability. This work provides an extensive overview of methods to measure the explainability and interpretability of machine learning results. This work addresses the challenges posed by closed-box models, which lack transparency in their decision-making processes and evaluates techniques used to make these models more understandable. Through the application of open-box models, like symbolic regression, we demonstrate that it is possible to achieve high interpretability without sacrificing model performance. Additionally, we highlight the necessity for robust and unified evaluation metrics for explainability and interpretability, evaluating complexity and fidelity scores as a comprehensive measure. A variety of closed-box models have been trained and evaluated in terms of performance, and explainability including, artificial neural networks, support vector regression, and random forest regression. Additionally, we trained symbolic regression models with different levels of complexity, which were evaluated regarding their performance and interpretability. Our results underscore the importance of developing methodologies that balance complexity and interpretability, advocating for further research into explainable artificial intelligence frameworks, particularly those incorporating genetic programming. This work aims to contribute to the advancement of responsible and transparent artificial intelligence systems.

E. Mayrhuber (✉)

Bioinformatics Research Group, PLFDoc, University of Applied Sciences Upper Austria, Hagenberg, Austria

e-mail: elisabeth.mayrhuber@fh-ooe.at

Computer Vision Lab, TU Wien, Vienna, Austria

B. Burlacu

Heuristic and Evolutionary Algorithms Laboratory, University of Applied Sciences Upper Austria, Hagenberg, Austria

e-mail: bogdan.burlacu@fh-ooe.at

S. M. Winkler

Department of Medical and Bioinformatics, University of Applied Sciences Upper Austria, Hagenberg, Austria

e-mail: stephan.winkler@fh-ooe.at

18.1 Introduction

Artificial Intelligence (AI) has become an integral part of numerous sectors, driving innovation and growth. However, as AI models become increasingly complex, understanding their decision-making process has emerged as a significant challenge. This opacity of AI models, often referred to as the closed-box (also known as black-box) problem, has led to the rise of Explainable Artificial Intelligence (XAI) [12].

A lot of surveys have been conducted in the field of XAI in order to identify methods and challenges [11, 17, 23, 28]. Not only the requirements for XAI [4, 24, 45] have been the focus of research, but also explainability is an important topic in all disciplines of machine learning [6, 8]. For example, Ding et al. [11] discuss the evolution of XAI, highlighting its importance in making AI decisions transparent and understandable to humans. They address the closed-box nature of AI algorithms, which hides their internal logic and decision-making processes, posing ethical and practical challenges. Further, the lack of consensus on evaluation metrics for XAI explanations, emphasizing the need for technical demonstration of interpretability benefits, is discussed. These findings underscore the necessity for XAI in achieving responsible and trustworthy AI, while also indicating the need for further methodological developments and regulatory frameworks.

Evolutionary computation (EC) offers techniques with high potential for interpretable machine learning, as many researchers have shown in the last few years [5, 29, 48]. EC methods can be used either to build intrinsic interpretability [36, 43] or to explain closed-box methods [14, 15]. The difference between closed-box and open-box (also known as white-box) models are shown in Fig. 18.1. Hu [18] highlights genetic programming (GP) as a potent tool for interpretable learning because it can evolve symbolic representations that capture complex relationships in data, making it easier for humans to understand. The author introduced the SMILE framework, which uses linear GP to develop interpretable models for metabolomics data, particularly focusing on Alzheimer's disease. SMILE helps identify key metabolites and their interactions, providing insights into the underlying biological processes of diseases. A web interface is developed for SMILE, allowing users to analyze, interpret, and visualize the results, enhancing the accessibility and usability of the interpretative models.

The work of Zhou and Hu [49] highlights the potential of EC as a powerful tool for optimization and learning that can significantly contribute to the field of XAI. Various current techniques for explaining machine learning models are reviewed, including data visualization, feature selection, and engineering, as well as local and global explanations. The publication discusses open challenges in XAI, such as scalability and incorporation of domain knowledge, and suggests opportunities for future research using EC to develop more transparent and accountable machine learning models. It emphasizes the importance of XAI in high-stakes applications like healthcare and finance, and the potential of EC to improve the interpretability and trustworthiness of AI models. The paper advocates for further exploration of

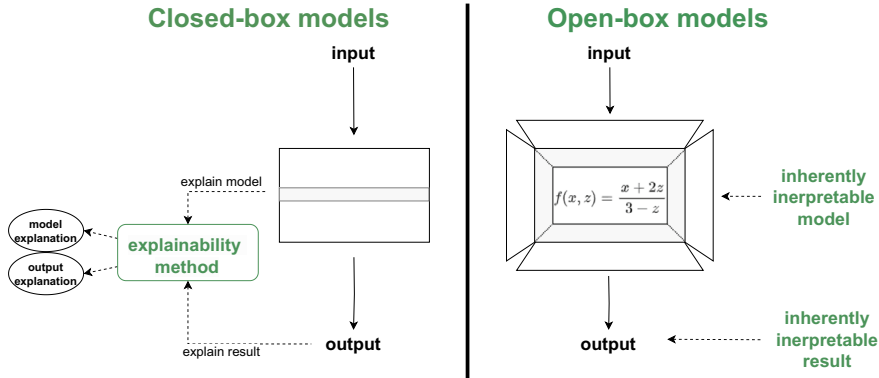


Fig. 18.1 When using closed-box models (left) explainability methods are needed to explain the model internals and results (explainability). On the other hand, open-box models (right) can be understood by looking at the internals of the model (interpretability)

evolutionary methods to enhance the explainability of complex machine learning models.

We present a comprehensive review of the current state of XAI, with a particular focus on the use of GP. It aims to compare the interpretability achievable through GP with that of closed-box models using post hoc explainability methods. The discussion is structured around three key topics: interpretability versus explainability, GP versus closed-box models, and evaluation methods for explainability. The first theme distinguishes between interpretability and explainability, two concepts that are often conflated in XAI literature. The second theme compares GP, a transparent model, with closed-box models, examining the strengths and weaknesses of each in terms of explainability. The third theme investigates how explainability is measured and evaluated, a critical aspect of XAI research. Through a comprehensive analysis of numerous papers and literature reviews, this paper aims to provide a thorough understanding of the current possibilities in XAI using GP and closed-box models. It also seeks to contribute to the ongoing discourse in the AI community about the importance of explainability and how it can be effectively measured and evaluated.

The methodology of this research is twofold. Firstly, it involves a comprehensive review of recent publications in the rapidly evolving field of XAI with a particular focus on the role of GP. Secondly, it aims to identify potential gaps in the current body of knowledge, thereby highlighting areas where future research could be beneficial. In order to gather the most relevant and recent publications in the field of XAI and GP, a series of search terms were carefully defined and used. These terms were designed to capture the breadth of the field while also focusing on specific areas of interest. The following search terms were used:

- (“Symbolic regression” OR “genetic programming”) AND “Explainable AI”: This query was designed to capture publications that discuss the intersection of symbolic regression or genetic programming with XAI.

- “Genetic programming” AND “Explainable AI” AND “methods”: This query focused on methods in the intersection of GP and XAI.
- “Artificial Intelligence” AND “Explainability” AND “methods”: This broader query returned numerous results, indicating a rich body of work on explainability methods in AI.
- (“Symbolic regression” OR “genetic programming”) AND “Explainable AI” AND (“AutoML” OR “Automated Machine Learning”): This query aimed to find works discussing the application of AutoML in the context of symbolic regression, genetic programming, and XAI.

These queries were executed on eight scientific databases, which are listed below to ensure a comprehensive search of the literature:

- ACM Digital Library¹
- IEEEExplore²
- IET Digital Library³
- Nature⁴
- PubMed⁵
- Science Direct⁶
- Springer Link⁷
- Wiley Online Library.⁸

From the total number of results, 227 publications were selected based on their relevance to the research topic. Of these, 56 were subsequently excluded as they were deemed not relevant to this research, and 28 were not accessible due to not being Open Access. The final selection consisted of 143 publications. Among these, 50 publications describe the application of XAI methods to recent problems, while the remainder are either literature reviews or describe novel methodologies in the sector.

18.2 Explainability Versus Interpretability

In some contexts, the terms “explainability” and “interpretability” are used interchangeably, as observed in [29]. However, other literature, including this work,

¹ <https://dl.acm.org/>.

² <https://ieeexplore.ieee.org/>.

³ <https://digital-library.theiet.org/>.

⁴ <https://www.nature.com/>.

⁵ <https://pubmed.ncbi.nlm.nih.gov/>.

⁶ <https://www.sciencedirect.com/>.

⁷ <https://link.springer.com/>.

⁸ <https://onlinelibrary.wiley.com/>.

treats them as distinct concepts [18]. Interpretability means that the internal of a open-box model can be understood by observing a model's structure and therefore understanding why a model is making a certain decision. At some point, (intrinsic) interpretability reaches its limits with growing model complexity. Some works claim that increased model performance comes with a certain lack of interpretability [29]. To create trustworthy machine learning applications, there is a need to improve model performance and at the same time make model decisions explainable [35].

Explainability means that decisions of closed-box models can be explained with the help of methods such as SHAP [25] or LIME [37]. GP can also be used to generate explanations for closed-box models, where GP is used to approximate a model decision based on its results and then use the internals of the GP model as a global explanation of a closed-box model [29]. This difference of explainability and interpretability is shown in Fig. 18.1.

Explainability can be classified into two types:

- Global post hoc explainability
- Local post hoc explainability.

These classes of explainability will be explained below with a focus on the role of EC in it.

18.2.1 Global Post Hoc Explainability

Global post hoc explainability means that the structure of a closed-box model is explained after training. Therefore, different methods can be used to create transparent surrogate models to interpret a closed-box model. When using GP to interpret a closed-box model post hoc, a dataset is generated from the model that should be explained. Then a GP model is trained on this dataset in order to get an interpretable model explaining the predictions [14]. In this case, there is again the limitation of model complexity, otherwise, the predictions cannot be interpreted [29]. Also, partial dependence plots are a way of visualizing the behavior of a model, by showing the effect of certain features on the output of a model. For more complex SR models also explainability methods can be used to enhance the interpretability, for example, by determining feature importance using partial effects [3].

18.2.2 Local Post Hoc Explainability

In local post hoc explainability, single predictions are explained instead of the whole model in general. Apart from SHAP and LIME which are widely known, local explainability methods, some EC approaches have also been researched. Also, technologies like large language models are used to create human-readable explanations of complex, not easily interpretable models. Maddigan et al. [26] try to explain

their results retrieved from GP using large language models and therefore to get user-centric explanations of the models' internal structure. The work of Maddigan et al. introduces GP4NLDR, an XAI dashboard that combines state-of-the-art GP with a Large Language Model (LLM)-powered chatbot to provide user-centered explanations of GP-based non-linear dimensionality reduction processes. The system demonstrated the ability to provide intuitive narratives on high-dimensional data reduction, emphasizing the importance of prompt engineering in eliciting accurate responses from LLMs. The research showcases the potential of LLMs, like those used in ChatGPT, in enhancing the explainability of GP algorithms, addressing considerations around data privacy, hallucinatory outputs, and the rapid advancements in generative AI. The methodology used involves a web-based dashboard with an LLM-powered chatbot, and the results indicate a significant advancement in making models resulting from GP more explainable and accessible.

18.3 How to Measure Explainability and Interpretability?

One open research topic in the field of XAI is evaluation metrics. A lot of work has been done in this direction. Still, there is a need for unified metrics where interpretability and explainability can be measured in models [21, 27, 30, 42].

18.3.1 *Evaluation Metrics for Explainability*

Makridis et al. [27] proposed a unified multidimensional explainability metric for AI models, focusing on fidelity, simplicity, and stability. They introduced a benchmarking framework to evaluate XAI methods like LIME and SHAP across multiple datasets. The framework includes the creation of an offline knowledge base with explainability scores for registered models, aiding in context-dependent evaluation. This work emphasizes the importance of a comprehensive metric that captures the quality and efficiency of AI model trustworthiness, supporting the development of transparent and trustworthy AI systems.

Kadir et al. [21] reviewed various methods and metrics that can be used to evaluate explainability. These methods include different types of metrics, like fidelity, human-centric approaches, and complexity metrics of numerical approximations. Fidelity evaluates how accurately explanations reflect the actual decision-making process of a model. Human-centric metrics assess the usefulness of explanations to humans. This can include surveys measuring satisfaction with explanations. Also, task performance metrics are a type of user-centric metric that assesses how explanations help users fulfill certain tasks. Complexity metrics evaluate the complexity of explanations or inherently interpretable models and numerical approximations use numerical methods to approximate and assess the quality of explanations.

To unify the evaluation of explanations, Nauta et al. [30] proposed 12 properties to assess explanation quality. The so-called Co-12 properties are classified into three categories, which are

- Content related
 - Correctness: Accuracy of the explanation in representing the model's decision process.
 - Completeness: The extent to which the explanation covers all relevant aspects of the model's decision.
 - Consistency: Stability of explanations across different instances and similar models.
 - Continuity: Consistency of the explanation across continuous input changes.
 - Contrastivity: Ability of the explanation to distinguish between different outcomes.
 - Covariate complexity: Complexity of feature interaction in the explanation.
- Presentation related
 - Compactness: Simplicity and size of the explanation.
 - Composition: Describes the organization of the explanation and the presentation format.
 - Confidence: Confidence level of the model in its predictions and corresponding explanations.
- User related
 - Context: Relevance of the explanation to the specific context or task.
 - Coherence: Describes how accordant the explanation is with prior knowledge.
 - Controllability: Degree to which users can interact with the explanation to gain better insights.

Speith and Langer [41, 42] introduced a classification approach for evaluation metrics. This classification approach includes three types of evaluation types: explanatory information, understanding, and desiderata. Explanatory information evaluation metrics focus on the quality of the explanatory information which is provided by the explanation. Understanding evaluation metrics aims to assess how well the explanation facilitates understanding among users, while desiderata evaluation metrics evaluate the satisfaction of social desiderata, like fairness and transparency. Therefore, Fidelity would be an evaluation metric for explanatory information. Each category comes with a different set of advantages and disadvantages. Explanatory information evaluation metrics are critical for understanding the system's behavior but may not fully capture the social impact. Therefore, the authors motivate to use multiple evaluation metrics to assess explanation quality.

Nneke et al. [31] used complexity measures to quantify explanations that were generated using SHAP. These complexity measures are intended to act as proxies for the explainability of closed-box models. Several complexity metrics, like linear

correlation and monotonicity. The authors proposed calculating a feature importance-weighted average of these complexity measures across the SHAP explanations for each feature to provide a single complexity score for a model. They observed a trade-off between the complexity of the explanations and model accuracy, indicating that more accurate models tend to have more complex explanations.

Research has also been conducted in the field of explanation representation, such as Kadir et al. [20] and Zöller et al. [50]. In [20], the authors introduced a user interface designed to enhance the understanding and evaluation of explanations generated by image-based deep-learning classifiers. The tool allows users to upload models and images, generate saliency maps, and visualize perturbation curves to assess the faithfulness of these explanations. By removing the most and least relevant features from input images, users can observe changes in model predictions, aiding in debugging and bias detection.

To evaluate explainability, Velmurugan et al. [44] used a fidelity score that consists of precision and recall which can be calculated using Eqs. (18.1) and (18.2). Recall (Eq. 18.2) measures completeness by calculating the proportion of true features captured by the explanation, and precision (Eq. 18.1) measures soundness by evaluating the proportion of relevant features in the explanation.

$$P = \frac{|F_{\text{SR}} \cap F_{\text{Explanation}}|}{F_{\text{SR}}} \quad (18.1)$$

$$R = \frac{|F_{\text{SR}} \cap F_{\text{Explanation}}|}{F_{\text{Explanation}}} \quad (18.2)$$

18.3.2 Evaluation Metrics for Interpretability

Similar to explainability, also the evaluation of interpretability is very subjective. A model can seem interpretable to user A while not being interpretable to user B. Interpretable models can also be evaluated according to Nauta et al. [30]. One way to assess interpretability without user input is to calculate complexity metrics. An example for such a complexity metric was introduced by Kommenda et al. [22]. To calculate the complexity, there are certain rules which need to be applied depending on the type of arithmetic symbol. For addition and subtraction, the calculation rule is shown in Eq. (18.3). In contrast, the rule for multiplication and division is shown in Eq. (18.4). The downside of this metric is that it is unbound, meaning it can grow very quickly.

$$C = \sum_{i=0}^n \text{Complexity}(C_i) \quad (18.3)$$

$$C = \prod_{i=0}^n \text{Complexity}(C_i) + 1 \quad (18.4)$$

Brotto Rebuli et al. [7] highlighted the potential of using structural complexity metrics to generate simpler and potentially more interpretable models in GP. However, they also showed that interpretability is not guaranteed by reducing model complexity alone. The effectiveness of these metrics varies across different datasets, and further research is needed to refine these approaches for better generalization and interpretability.

Many of the named works use complexity metrics as well as fidelity scores to assess the explanation complexity. Therefore, in this work, we also use complexity to assess the interpretability of, open-box, SR models. For closed-box models, like neural networks and support vector regression, we are using a fidelity score to assess explanations for the models which are generated using SHAP.

18.4 Evaluation of Explainability and Interpretability

The high variety in areas where XAI is used shows the immense desire to understand predictions given by ML models. Especially in highly critical areas, like healthcare [10, 16] and the legal sector, decisions made by models need to be understood by humans in order to prevent wrong decisions. One example where XAI is often used within healthcare is in the diagnosis of Parkinson's disease [9, 19, 33]. For example, Cavaliere et al. [9] discussed the use of grammar-based XAI for diagnosing Parkinson's Disease (PD), emphasizing the need for transparency in medical decision-making. They use grammatical evolution (GE) that generates explicit classification rules for PD diagnosis through handwriting analysis, making the process interpretable for physicians. The research evaluates GE on the HandPD dataset, which contains handwriting samples from healthy subjects and PD patients, aiming to design a non-invasive and quick diagnostic tool. The study finds that GE provides comparable performance to other Machine Learning models while offering better explainability, and suggests future work to collect a more balanced dataset for further evaluation.

In industry, XAI is often used by leveraging SHAP or LIME to understand the importance of certain features within a prediction [19, 39, 47]. Also, the field of precision livestock farming aims for interpretable models, and therefore GP was used in the past [2], as well as genetic algorithms [1, 34].

This study aims to compare model performance, explainability, and interpretability of different ML techniques. In order to do so five different regression models were trained on three different datasets.

The used datasets are

- California housing dataset [32].
- Diabetes dataset [13].
- A generated dataset where 400 instances with eight features were generated using a Gaussian distribution and a noise of one (standard deviation of the distribution).

The California housing dataset includes eight features, including the house age, number of bedrooms, and property location. The goal is to predict the value of the houses. The diabetes dataset includes 442 instances with 10 features including BMI, age, average blood pressure of patients, and also some laboratory values. Four different methods were used to train the models, as listed below:

- Symbolic regression with genetic programming
 - SR_small (maximum generations: 1000, maximum selection pressure: 100, population size: 100, maximum symbolic expression tree depth: 8, maximum symbolic expression tree length: 50, optimizer: offspring selection evolution strategy, allowed functions: +, −, *, /, exp, log).
 - SR_complex: (population size: 1000, elitism rate: 0.05, crossover probability: 0.9, mutation probability: 0.2, maximum generations: 1000, no size limitation, allowed functions: full EC-KitY toolbox except OR).
- Artificial neural network (number of hidden layers: 100, activation function: relu, maximum iterations: 200, constant learning rate).
- Support vector regression (kernel: rbf, degree: 3, C: 1, gamma: scale).
- Random forest regression (number of estimators: 100, max depth: None, criterion: squared error).

To train the symbolic regression, EC-KitY [40] and HeuristicLab [46] were used. Two SR models were trained to compare different complexity levels, in the following they will be referred to as SR_complex and SR_small. For all the other models, scikit-learn was utilized for model training. All models were trained using a threefold cross-validation to ensure reliable results. Table 18.1 shows a comprehensive overview of the results and will be further explained in the following.

18.4.1 Evaluation of the Interpretability of Open-Box Models

The resulting models for the SR_small for all datasets can be seen in Eqs. (18.5)–(18.7), where Eq. (18.5) is the model for the California housing dataset, Eq. (18.6) is the model for the diabetes dataset, and Eq. (18.7) shows the model for the generated dataset. To evaluate the interpretability of these models, a complexity metric, introduced by Kommenda et al. [22], is used. The complexity scores for these models can be found in Table 18.1 and an example for the calculation can be seen in Fig. 18.2, where the calculation for the SR_small model of the California housing dataset is visualized.

Table 18.1 This table shows the performance of each model where R^2 shows the coefficient of determination and MSE the mean squared error for each model. Further, the results of the explainability and interoperability analysis are shown, where the precision and recall are represented with respect to the complex and small SR models for each dataset and closed-box model. The complexity score follows the definition of Kommenda et al. [22]. Size and depth are not the same for each model type: The size of a RF model is the total number of nodes of all trees included in the random forest and the depth is the maximum depth of the deepest tree. For SVR, the size is the total number of support vectors in the system and for ANNs the size is again the total number of nodes and the depth is the number of layers included in the network

Dataset	Model	Performance			Explainability				Interpretability		
		R^2	MSE		Fidelity (SR_complex)		Fidelity (SR_small)		Complexity	Size*	Depth*
			Mean	Std	Mean	Std	Precision	Recall			
Generated	RF	0.729	0.018	9214	904	1	1	1	0.75	33878	18
	SVR	0.028	0.01	32965	1534	0.997	1	0.998	0.751	2144	–
	ANN	0.13	0.005	29501	1382	1	1	1	0.75	900	3
	SR_complex	0.982	0.005	590.9	123.1	–	–	–	–	> 1000	20
	SR_small	0.833	0.049	3125	1625	–	–	–	34	23	10
California Housing	RF	0.791	0.006	0.278	0.006	0.875	1	0.977	0.568	1684600	37
	SVR	–0.0384	0.001	1.382	0.023	0.249	0.964	0.09	0.146	101528	–
	ANN	0.271	0.25	0.973	0.342	0.846	1	0.967	0.571	900	3
	SR_complex	0.577	0.029	0.564	0.047	–	–	–	–	> 1000	12
	SR_small	0.246	0.024	0.747	0.063	–	–	–	28	11	7
Diabetes	RF	0.372	0.083	3646	353.8	0.99	1	1	0.2	36512	21
	SVR	0.0608	0.036	5577	372.9	0.99	1	0.997	0.2	2960	–
	ANN	–3.659	0.421	27687	2229	0.98	1	0.989	0.2	1100	3
	SR_complex	0.182	0.239	4793	1179	–	–	–	–	> 1000	16
	SR_small	0.131	0.044	3664	701.2	–	–	–	8	17	9

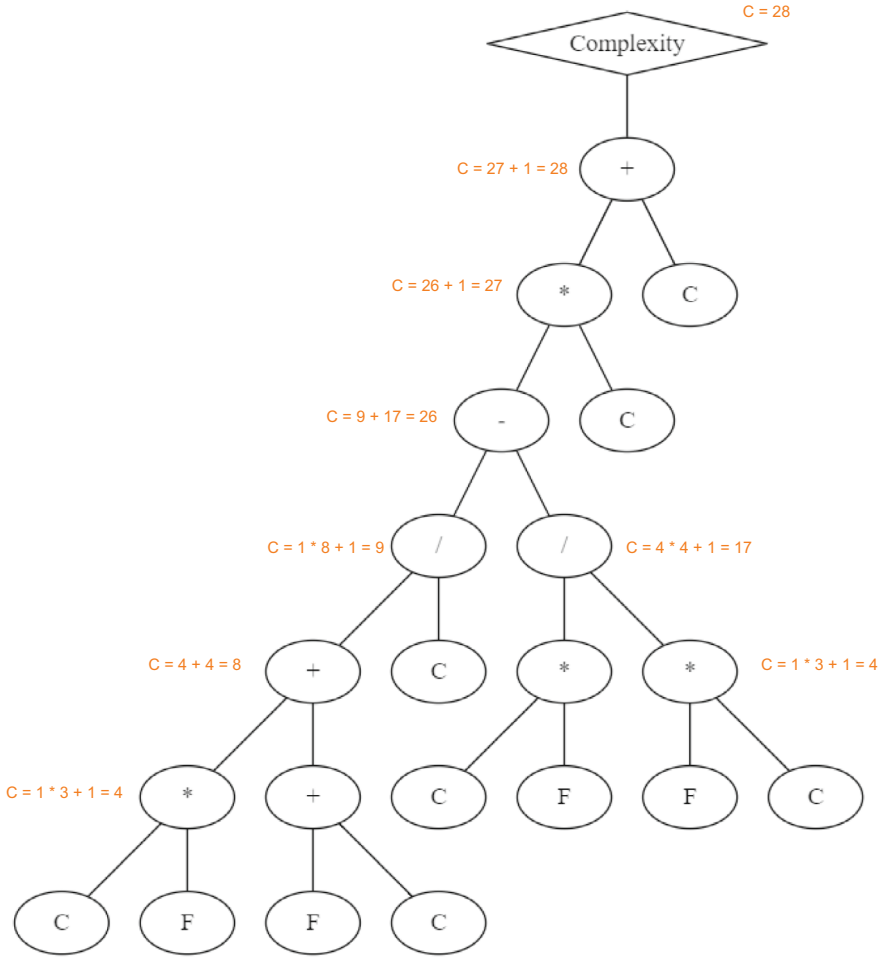


Fig. 18.2 This tree shows the complexity calculation for the SR_small model and the California housing dataset (Eq. 18.5) according to Kommenda et al. [22]. The initial values for constants are 1 and for features 2 but multiplied with a constant in the formulas the initial values for features result in 3

$$f(\text{Lon}, \text{Lat}, \text{MedInc}, \text{AvgOcc}) = -1.785 + \left(-1.15 * \left(\frac{0.4 * \text{Lon} + 0.6 * \text{Lat}}{2} - \frac{0.7 * \text{MedInc}}{0.8 * \text{AvgOcc}} \right) \right) \quad (18.5)$$

$$f(\text{BMI}, s_6) = -1, 15 * 10^4 + -617, 3 * (1, 4 * \text{BMI} - (-18, 8 - 0, 47 * s_6)) \quad (18.6)$$

$$f(f_2, f_3, f_4, f_5, f_6, f_8) =$$

$$-362 - 54,7 * (1,68 * f2 - \frac{-17,4 - 1,14 * f5 - 0,997 * f8}{-0,235 * f3 + 2,65} - f5 + f6 + f4 + f8) \quad (18.7)$$

The results for this complexity analysis can be found in Table 18.1. This complexity score cannot be calculated for the closed-box models. To still be able to compare some kind of complexity for all the models, the model sizes are outlined in Table 18.1. Since the internals of these models are not the same the interpretation must be done with caution. For the RF models, the size is the total number of nodes of all the decision trees contained in the random forest and the depth is the maximum depth of the deepest tree. For the SVR no depth can be calculated but the size is the total number of support vectors in the model. The model size for the ANN models is the total number of nodes in the network and the depth is the number of layers in the network. For the SR models, the size is the number of nodes and the depth is the maximum depth of the tree. When looking at the results in Table 18.1 it is clear that the SR models are the ones with the lowest complexity compared to the closed-box models and therefore easier inherently interpretable.

18.4.2 Evaluation of the Explainability of Closed-Box Models

For this analysis, we used SHAP to explain the closed-box models. With SHAP global and local explanations can be generated. Figure 18.3 shows the global explanation for the RF model on the California housing dataset and Fig. 18.4 shows a local explanation for one instance for the same model and dataset.

SHAP [25] works by assigning each feature an importance value for a particular prediction, based on Shapley values [38]. It calculates the contribution of each feature by considering all possible combinations of features, ensuring a fair and consistent allocation of feature importance. This approach provides a unified measure of feature importance across different types of models, making it easier to interpret complex model predictions.

In both, the global and local explanations (Figs. 18.3 and 18.4), the feature “Med-Inc”, which is the median income in a block group, has the highest impact on the result. This is followed by the latitude and longitude of the property. When comparing the SHAP explanations to the open-box models, it can be seen that the four most important features are the ones used by the SR_small (Eq. 18.5).

The fidelity scores, which can be found in Table 18.1, measure how well the explanations of a machine learning model’s decisions align with the decision-making process of the open-box model. The used method in [44] is adapted from Ribeiro et al. [37] who evaluated explanations generated with LIME.

Precision and recall for the fidelity (Eqs. 18.1 and 18.2) are calculated at the instance level and are averaged over the whole dataset for a comprehensive evaluation. This analysis has been done for all closed-box models regarding both open-box models (SR_small and SR_complex) as shown in Table 18.1. When looking at the

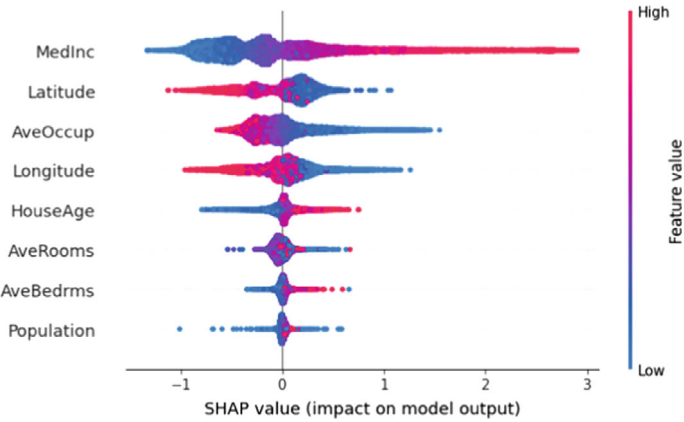


Fig. 18.3 Global explanation using SHAP, showing that the median income per block group has the highest impact on the result. The color indicated the feature value how these values influence the model output (SHAP values). The features are sorted by their impact on the model output

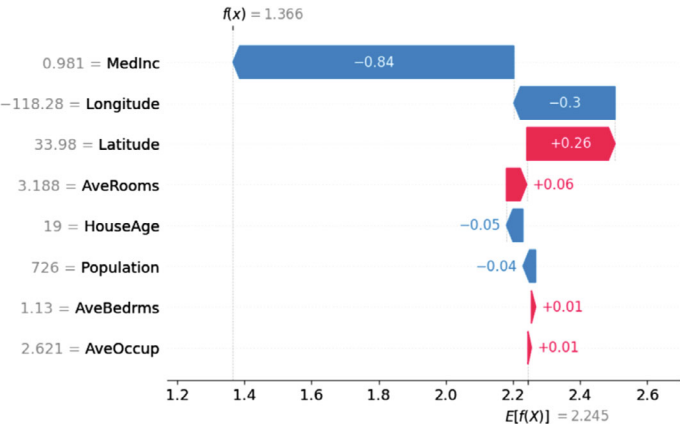


Fig. 18.4 Local explanation for one instance using SHAP, showing that the median income per block group has the highest impact on the result

results it can be seen that the explanations of the closed-box models tend to use more features than the SR_small model. The SR_complex model uses all the available features for the prediction and since precision and recall are mostly near 1 it can be said that the explanations of the closed-box models also use all the available features for each dataset. One exception is the SVR for the California housing dataset since the precision for the SR_complex model is 0.249 and the precision for the SR_small is 0.09, which means that the explanation for the SVR does not use all available features but the features that are used are very different than the ones in the SR models. For the generated dataset, the two symbolic regression models achieve the best results when looking at the R^2 score and the mean squared error (MSE). For the other

datasets, the RF models perform slightly better than the SR_small and SR_complex but the RF models have a higher complexity than the SR models.

18.5 Conclusion

This work gives an overview of explainability and interpretability and outlines the need for explainable machine learning methods. Further, it evaluates the explainability of different closed-box models as well as the interpretability of certain open-box models. It shows that symbolic regression is a strong method to create compact, interpretable models, with a comparable performance to more complex models. This means that there is not necessarily a trade-off between performance and interpretability. For complex problems where small, inherently interpretable models are not able to create satisfying results, presentation methods need to be developed to make more complex models easily understandable. The possibilities to evaluate explainability and interpretability in ML models are currently limited and therefore robust and unified evaluation metrics are highly needed and state an open research question. Also, a lot of research needs to be done in the field of explanation representation to make the results of interpretable models and explanations better understandable for human domain experts. In this sense, the combination of AutoML and XAI especially XAI by genetic programming should be further investigated in order to get complete systems that create interpretable predictions.

Acknowledgements This research was funded in whole or in part by the Austrian Science Fund (FWF) [<https://doi.org/10.55776/DFH34>]. For open access purposes, the author has applied a CC BY public copyright license to any author-accepted manuscript version arising from this submission.

References

1. Abbona, F., Vanneschi, L., Bona, M., Giacobini, M.: A GP approach for precision farming. In: 2020 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8 (2020). <https://doi.org/10.1109/CEC48606.2020.9185637>
2. Abbona, F., Vanneschi, L., Bona, M., Giacobini, M.: Towards modelling beef cattle management with Genetic Programming. *Livest. Sci.* **241**, 104,205 (2020). <https://doi.org/10.1016/j.livsci.2020.104205>. <https://www.sciencedirect.com/science/article/pii/S1871141320302481>
3. Aldeia, G.S.I., De França, F.O.: Measuring feature importance of symbolic regression models using partial effects. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 750–758. ACM, Lille France (2021). <https://doi.org/10.1145/3449639.3459302>. <https://dl.acm.org/doi/10.1145/3449639.3459302>
4. Ali, S., Abuhmed, T., El-Sappagh, S., Muhammad, K., Alonso-Moral, J.M., Confalonieri, R., Guidotti, R., Ser, J.D., Díaz-Rodríguez, N., Herrera, F.: Explainable Artificial Intelligence (XAI): what we know and what is left to attain trustworthy artificial intelligence. *Inf. Fusion* **99**, 101,805 (2023). <https://doi.org/10.1016/j.inffus.2023.101805>. <https://www.sciencedirect.com/science/article/pii/S1566253523001148>

5. Bacardit, J., Brownlee, A.E.I., Cagnoni, S., Iacca, G., McCall, J., Walker, D.: The intersection of evolutionary computation and explainable AI. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 1757–1762. ACM, Boston Massachusetts (2022). <https://doi.org/10.1145/3520304.3533974>. <https://dl.acm.org/doi/10.1145/3520304.3533974>
6. Bekkemoen, Y.: Explainable reinforcement learning (XRL): a systematic literature review and taxonomy. *Mach. Learn.* **113**(1), 355–441 (2024). <https://doi.org/10.1007/s10994-023-06479-7>. <https://link.springer.com/10.1007/s10994-023-06479-7>
7. Brotto Rebuli, K., Giacobini, M., Silva, S., Vanneschi, L.: A comparison of structural complexity metrics for explainable genetic programming. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation, pp. 539–542. ACM, Lisbon Portugal (2023). <https://doi.org/10.1145/3583133.3590595>. <https://dl.acm.org/doi/10.1145/3583133.3590595>
8. Burkart, N., Huber, M.F.: A survey on the explainability of supervised machine learning. *JAIR* **70**, 245–317 (2021). <https://doi.org/10.1613/jair.1.12228>. <https://jair.org/index.php/jair/article/view/12228>
9. Cavaliere, F., Cioppa, A.D., Marcelli, A., Parziale, A., Senatore, R.: Parkinson’s disease diagnosis: towards grammar-based explainable artificial intelligence. In: 2020 IEEE Symposium on Computers and Communications (ISCC), pp. 1–6 (2020). <https://doi.org/10.1109/ISCC50000.2020.9219616>. ISSN: 2642-7389
10. Di Martino, F., Delmastro, F.: Explainable AI for clinical and remote health applications: a survey on tabular and time series data. *Artif. Intell. Rev.* **56**(6), 5261–5315 (2023). <https://doi.org/10.1007/s10462-022-10304-3>. <https://link.springer.com/10.1007/s10462-022-10304-3>
11. Ding, W., Abdel-Basset, M., Hawash, H., Ali, A.M.: Explainability of artificial intelligence methods, applications and challenges: a comprehensive survey. *Inf. Sci.* **615**, 238–292 (2022). <https://doi.org/10.1016/j.ins.2022.10.013>. <https://www.sciencedirect.com/science/article/pii/S002002552201132X>
12. Dwivedi, R., Dave, D., Naik, H., Singhal, S., Omer, R., Patel, P., Qian, B., Wen, Z., Shah, T., Morgan, G., Ranjan, R.: Explainable AI (XAI): core ideas, techniques, and solutions. *ACM Comput. Surv.* **55**(9), 1–33 (2023). <https://doi.org/10.1145/3561048>. <https://dl.acm.org/doi/10.1145/3561048>
13. Efron, B., Hastie, T., Johnstone, I., Tibshirani, R.: Least angle regression. *Ann. Stat.* **32**(2) (2004). <https://doi.org/10.1214/009053604000000067>. <http://arxiv.org/abs/math/0406456>. [ArXiv:math/0406456](https://arxiv.org/abs/math/0406456)
14. Evans, B.P., Xue, B., Zhang, M.: What’s inside the black-box?: a genetic programming method for interpreting complex machine learning models. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1012–1020. ACM, Prague Czech Republic (2019). <https://doi.org/10.1145/3321707.3321726>. <https://dl.acm.org/doi/10.1145/3321707.3321726>
15. Ferreira, L.A., Guimarães, F.G., Silva, R.: Applying genetic programming to improve interpretability in machine learning models. In: 2020 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8 (2020). <https://doi.org/10.1109/CEC48606.2020.9185620>
16. González-Nóvoa, J.A., Campanioni, S., Busto, L., Fariña, J., Rodríguez-Andina, J.J., Vila, D., Íñiguez, A., Veiga, C.: Improving intensive care unit early readmission prediction using optimized and explainable machine learning. *IJERPH* **20**(4), 3455 (2023). <https://doi.org/10.3390/ijerph20043455>. <https://www.mdpi.com/1660-4601/20/4/3455>
17. Hassija, V., Chamola, V., Mahapatra, A., Singal, A., Goel, D., Huang, K., Scardapane, S., Spinelli, I., Mahmud, M., Hussain, A.: Interpreting black-box models: a review on explainable artificial intelligence. *Cogn. Comput.* **16**(1), 45–74 (2024). <https://doi.org/10.1007/s12559-023-10179-8>. <https://link.springer.com/10.1007/s12559-023-10179-8>
18. Hu, T.: Genetic programming for interpretable and explainable machine learning. In: Trujillo, L., Winkler, S.M., Silva, S., Banzhaf, W. (eds.) *Genetic Programming Theory and Practice XIX*, pp. 81–90. Springer Nature Singapore, Singapore (2023). https://doi.org/10.1007/978-981-19-8460-0_4. https://link.springer.com/10.1007/978-981-19-8460-0_4
19. Junaid, M., Ali, S., Eid, F., El-Sappagh, S., Abuhmed, T.: Explainable machine learning models based on multimodal time-series data for the early detection of Parkinson’s disease. *Comput. Methods Programs Biomed.* **234**, 107,495 (2023). <https://doi.org/10.1016/j.cmpb.2023.107495>. <https://www.sciencedirect.com/science/article/pii/S016926072300161X>

20. Kadir, M.A., Mohamed Selim, A., Barz, M., Sonntag, D.: A user interface for explaining machine learning model explanations. In: 28th International Conference on Intelligent User Interfaces, pp. 59–63. ACM, Sydney NSW Australia (2023). <https://doi.org/10.1145/3581754.3584131>
21. Kadir, M.A., Mosavi, A., Sonntag, D.: Evaluation metrics for XAI: a review, taxonomy, and practical applications. In: 2023 IEEE 27th International Conference on Intelligent Engineering Systems (INES), pp. 000,111–000,124 (2023). <https://doi.org/10.1109/INES59282.2023.10297629>. ISSN: 1543-9259
22. Kommenda, M., Kronberger, G., Affenzeller, M., Winkler, S.M., Burlacu, B.: Evolving Simple Symbolic Regression Models by Multi-Objective Genetic Programming, pp. 1–19. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-34223-8_1
23. Krenn, M., Pollice, R., Guo, S.Y., Aldeghi, M., Cervera-Lierta, A., Friederich, P., dos Passos Gomes, G., Häse, F., Jinich, A., Nigam, A., Yao, Z., Aspuru-Guzik, A.: On scientific understanding with artificial intelligence. *Nat. Rev. Phys.* **4**(12), 761–769 (2022). <https://doi.org/10.1038/s42254-022-00518-3>. <https://www.nature.com/articles/s42254-022-00518-3>
24. Liu, H., Wang, Y., Fan, W., Liu, X., Li, Y., Jain, S., Liu, Y., Jain, A., Tang, J.: Trustworthy AI: a computational perspective. *ACM Trans. Intell. Syst. Technol.* **14**(1), 1–59 (2023). <https://doi.org/10.1145/3546872>. <https://dl.acm.org/doi/10.1145/3546872>
25. Lundberg, S.M., Lee, S.I.: A unified approach to interpreting model predictions. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) *Advances in Neural Information Processing Systems*, vol. 30. Curran Associates, Inc. (2017). https://proceedings.neurips.cc/paper_files/paper/2017/file/8a20a8621978632d76c43dfd28b67767-Paper.pdf
26. Maddigan, P., Lensen, A., Xue, B.: Explaining Genetic Programming Trees using Large Language Models (2024). <http://arxiv.org/abs/2403.03397>. [ArXiv:2403.03397](https://arxiv.org/abs/2403.03397) [cs]
27. Makridis, G., Fatouros, G., Kiourtis, A., Kotios, D., Koukos, V., Kyriazis, D., Soldatos, J.: Towards a unified multidimensional explainability metric: evaluating trustworthiness in AI models. In: 2023 19th International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT), pp. 504–511 (2023). <https://doi.org/10.1109/DCOSS-IoT58021.2023.00084>. ISSN: 2325-2944
28. Marcinkevičs, R., Vogt, J.E.: Interpretable and explainable machine learning: a methods-centric overview with concrete examples. *WIREs Data Min. Knowl.* **13**(3), e1493 (2023). <https://doi.org/10.1002/widm.1493>. <https://wires.onlinelibrary.wiley.com/doi/10.1002/widm.1493>
29. Mei, Y., Chen, Q., Lensen, A., Xue, B., Zhang, M.: Explainable artificial intelligence by genetic programming: a survey. *IEEE Trans. Evol. Computat.* **27**(3), 621–641 (2023). <https://doi.org/10.1109/TEVC.2022.3225509>. <https://ieeexplore.ieee.org/document/9965435/>
30. Nauta, M., Trienes, J., Pathak, S., Nguyen, E., Peters, M., Schmitt, Y., Schlötterer, J., Van Keulen, M., Seifert, C.: From anecdotal evidence to quantitative evaluation methods: a systematic review on evaluating explainable AI. *ACM Comput. Surv.* **55**(13s), 1–42 (2023). <https://doi.org/10.1145/3583558>. <https://dl.acm.org/doi/10.1145/3583558>
31. Nneke, N., Lloyd, H., Crockett, K.: Quantification of explainability in black box models using complexity measures. In: 2023 15th International Conference on Innovations in Information Technology (IIT), pp. 85–92 (2023). <https://doi.org/10.1109/IIT59782.2023.10366490>. ISSN: 2473-2052
32. Pace, K., Barry, R.: Sparse spatial autoregressions. *Stat. Probab. Lett.* **33**, 291–297 (1997). [https://doi.org/10.1016/S0167-7152\(96\)00140-X](https://doi.org/10.1016/S0167-7152(96)00140-X)
33. Parziale, A., Senatore, R., Cioppa, A.D., Marcelli, A.: Cartesian genetic programming for diagnosis of Parkinson disease through handwriting analysis: performance vs. interpretability issues. *Artif. Intell. Med.* **111**, 101,984 (2021). <https://doi.org/10.1016/j.artmed.2020.101984>. <https://www.sciencedirect.com/science/article/pii/S0933365720312495>
34. Pazouki, E.: Optimizing an irrigation treatment using an evolutionary algorithm and a knowledge discovery framework based on Deep models. *Appl. Soft Comput.* **133**, 109,940 (2023). <https://doi.org/10.1016/j.asoc.2022.109940>. <https://www.sciencedirect.com/science/article/pii/S1568494622009899>

35. Petkovic, D.: It is not “Accuracy vs. Explainability”—we need both for trustworthy AI systems. *IEEE Trans. Technol. Soc.* **4**(1), 46–53 (2023). <https://doi.org/10.1109/TTS.2023.3239921>
36. Quade, M., Isele, T., Abel, M.: Machine learning control—explainable and analyzable methods. *Phys. D: Nonlinear Phenom.* **412**, 132,582 (2020). <https://doi.org/10.1016/j.physd.2020.132582>. <https://www.sciencedirect.com/science/article/pii/S0167278920300026>
37. Ribeiro, M.T., Singh, S., Guestrin, C.: “Why Should I Trust You?”: Explaining the Predictions of Any Classifier (2016). <https://doi.org/10.48550/ARXIV.1602.04938>. <https://arxiv.org/abs/1602.04938>. Publisher: [object Object] Version Number: 3
38. Roth, A.E.: Introduction to the Shapley Value, p. 1–28. Cambridge University Press (1988)
39. Salem, H., El-Hasnony, I.M., Kabeel, A.E., El-Said, E.M.S., Elzeki, O.M.: Deep learning model and classification explainability of renewable energy-driven membrane desalination system using evaporative cooler. *Alex. Eng. J.* **61**(12), 10007–10024 (2022). <https://doi.org/10.1016/j.aej.2022.03.050>. <https://www.sciencedirect.com/science/article/pii/S1110016822002241>
40. Sipper, M., Halperin, T., Tzruia, I., Elyasaf, A.: EC-KitY: evolutionary computation tool kit in Python with seamless machine learning integration. *SoftwareX* **22**, 101,381 (2023). <https://doi.org/10.1016/j.softx.2023.101381>. <https://linkinghub.elsevier.com/retrieve/pii/S2352711023000778>
41. Speith, T.: How to evaluate explainability?—a case for three criteria. In: 2022 IEEE 30th International Requirements Engineering Conference Workshops (REW), pp. 92–97 (2022). <https://doi.org/10.1109/REW56159.2022.00024>. ISSN: 2770-6834
42. Speith, T., Langer, M.: A new perspective on evaluation methods for explainable artificial intelligence (XAI). In: 2023 IEEE 31st International Requirements Engineering Conference Workshops (REW), pp. 325–331 (2023). <https://doi.org/10.1109/REW57809.2023.00061>. ISSN: 2770-6834
43. Tran, B., Sudusinghe, C., Nguyen, S., Alahakoon, D.: Building interpretable predictive models with context-aware evolutionary learning. *Appl. Soft Comput.* **132**, 109,854 (2023). <https://doi.org/10.1016/j.asoc.2022.109854>. <https://www.sciencedirect.com/science/article/pii/S1568494622009036>
44. Velmurugan, M., Ouyang, C., Moreira, C., Sindhgatta, R.: Developing a fidelity evaluation approach for interpretable machine learning (2021). *CoRR* **abs/2106.08492**
45. Vieira, C.P., Digiampietri, L.A.: Machine Learning post-hoc interpretability: a systematic mapping study. In: XVIII Brazilian Symposium on Information Systems, pp. 1–8. ACM, Curitiba Brazil (2022). <https://doi.org/10.1145/3535511.3535512>
46. Wagner, S., Affenzeller, M.: HeuristicLab: a generic and extensible optimization environment. In: Ribeiro, B., Albrecht, R.F., Dobnikar, A., Pearson, D.W., Steele, N.C. (eds.) *Adaptive and Natural Computing Algorithms*, pp. 538–541. Springer, Vienna (2005). https://doi.org/10.1007/3-211-27389-1_130. http://link.springer.com/10.1007/3-211-27389-1_130
47. Zheng, X., Xie, Y., Yang, X., Amin, M.N., Nazar, S., Khan, S.A., Althoe, F., Deifalla, A.F.: A data-driven approach to predict the compressive strength of alkali-activated materials and correlation of influencing parameters using SHapley Additive exPlanations (SHAP) analysis. *J. Mater. Res. Technol.* **25**, 4074–4093 (2023). <https://doi.org/10.1016/j.jmrt.2023.06.207>. <https://www.sciencedirect.com/science/article/pii/S2238785423014552>
48. Zhou, R., Bacardit, J., Brownlee, A., Cagnoni, S., Fyvie, M., Iacca, G., McCall, J., van Stein, N., Walker, D., Hu, T.: Evolutionary computation and explainable AI: a roadmap to transparent intelligent systems (2024). <https://arxiv.org/abs/2406.07811>
49. Zhou, R., Hu, T.: Evolutionary approaches to explainable machine learning. In: Banzhaf, W., Machado, P., Zhang, M. (eds.) *Handbook of Evolutionary Machine Learning*, pp. 487–506. Springer Nature Singapore, Singapore (2024). https://doi.org/10.1007/978-981-99-3814-8_16. https://link.springer.com/10.1007/978-981-99-3814-8_16. Series Title: Genetic and Evolutionary Computation
50. Zöller, M.A., Titov, W., Schlegel, T., Huber, M.F.: XAutoML: a visual analytics tool for understanding and validating automated machine learning. *ACM Trans. Interact. Intell. Syst.* **13**(4), 1–39 (2023). <https://doi.org/10.1145/3625240>. <https://dl.acm.org/doi/10.1145/3625240>

Chapter 19

Lexicase Selection Parameter Analysis: Varying Population Size and Test Case Redundancy with Diagnostic Metrics



Jose Guadalupe Hernandez, Anil Kumar Saini, and Jason H. Moore

Abstract Lexicase selection is a successful parent selection method in genetic programming that has outperformed other methods across multiple benchmark suites. Unlike other selection methods that require explicit parameters to function, such as tournament size in tournament selection, lexicase selection does not. However, if evolutionary parameters like population size and number of generations affect the effectiveness of a selection method, then lexicase's performance may also be impacted by these 'hidden' parameters. Here, we study how these hidden parameters affect lexicase's ability to exploit gradients and maintain specialists using diagnostic metrics from an existing benchmark suite that measures a selection scheme's ability to exploit and explore handcrafted search spaces. By varying the population size with a fixed evaluation budget, we show that smaller populations tend to have greater exploitation capabilities, whereas larger populations tend to maintain more specialists. We also consider the effect redundant test cases have on specialist maintenance, and find that high redundancy may hinder the ability to optimize and maintain specialists, even for larger populations. Ultimately, we highlight that the interaction between population size, evaluation budget, and test cases must be carefully considered for the characteristics of the problem being solved.

19.1 Introduction

Genetic programming has successfully solved problems from various domains such as program synthesis, automated machine learning, and multi-objective optimization. However, multiple components, procedures, and parameters must be tailored for a given problem to increase the likelihood of a successful genetic programming

J. G. Hernandez (✉) · A. K. Saini · J. H. Moore
Cedars-Sinai Medical Center, Los Angeles, CA, USA
e-mail: jose.hernandez8@cshs.org

A. K. Saini
e-mail: anil.saini@cshs.org

J. H. Moore
e-mail: jason.moore@csmc.edu

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2025
S. M. Winkler et al. (eds.), *Genetic Programming Theory and Practice XXI*, Genetic and Evolutionary Computation, https://doi.org/10.1007/978-981-96-0077-9_19

run. For example, decisions on solution representation, variation operators, selection methods, population size, and the number of generations need to be made before starting a genetic programming run. Additionally, each component or procedure may contain its own set of parameters to consider, such as determining the type of crossover (e.g., one-point, alternation, etc.). Ultimately, all the decisions made in designing a genetic programming system compound and influence the likelihood of success.

A variety of parent selection methods (selection schemes) exist within genetic programming to identify promising solutions to continue evolving. Some of these methods require an explicit set of parameters to operate. Tournament selection, for instance, requires a tournament size to identify parents, whereas other methods like fitness-proportionate selection and lexicase selection are *parameter-free*. In practice, selection methods are typically implemented as functions in programming parlance that access a population of evaluated individuals. Given the programming framing, all selection methods at a minimum require these two ‘hidden’ parameters: (1) population of evaluated solutions and (2) their performances. Both of these parameters influence a selection method: population size determines the number of potential options and performances determine the priority (quality) of a solution.

Redundancy in test cases can also affect the behavior of a parent selection method. For example, methods that aggregate performances will be biased toward redundant test cases, as they are overrepresented in the total sum. This bias can also occur within methods that do not directly aggregate performances, such as lexicase selection. In fact, duplicate test cases are more likely to appear early when the set of test cases is randomly shuffled. As such, lexicase will disproportionately prefer individuals who are best on the test cases with multiple duplicates. In such a case, more selection events (bigger population size, more generations, or both) are needed to offset this effect. In real-world problems, redundancy can present itself as duplicate cases in the test set, and as similar test cases such that the performances of solutions on these test cases would be highly correlated. In this work, we study the dynamics of lexicase with the first type of redundancy and provide insights for future research directions regarding the second type of redundancy.

The ability to intuitively understand how a parent selection method guides an evolutionary search is critical to understanding the problems a particular method is best suited for. As such, we use the DOSSIER suite from [11] to understand how population size affects lexicase selection’s ability to exploit gradients and maintain specialists given an evaluation budget. We found that smaller populations facilitate more exploitation, as smaller populations are given more generations to evolve and accumulate mutations. Conversely, we found that larger population sizes can maintain and optimize more specialists. However, larger population sizes may negatively impact specialist maintenance when redundant test cases are also used to identify parents. Ultimately, we illustrate the importance of carefully considering the population size, evaluation budget, and test cases when using lexicase selection.

19.2 Lexicase Selection Analysis

19.2.1 *Lexicase Selection*

Many parent selection methods, such as tournament and fitness proportionate selection, typically use the total error on specific test cases to identify parent solutions. Lexicase selection [7], however, considers the error on each test case separately. The following process is used to select one parent from the entire population. First, the set of test cases is randomly shuffled. Then, the individuals who perform the best on the first test case are kept in the pool of candidates and the rest are removed. The next test case in the sequence is considered and the same process is followed. We repeat these steps until only one individual remains in the candidate pool, or no test cases remain. In the latter case, we randomly choose an individual from the remaining candidates to become a parent.

Other variants of lexicase have also been proposed. For example, epsilon-lexicase selection [14] is used for problems where the fitness values are real-valued. In down-sampled lexicase and its variants [1, 9], for every selection event, only a subset of test cases is used to select a parent. While these variants of lexicase have proven themselves effective, we focus on the original lexicase selection in this work.

19.2.2 *Analyzing Performance*

Lexicase selection has been used successfully in a variety of evolutionary computation domains. Program synthesis is one such domain. It is the process of generating computer programs that satisfy a given set of constraints or examples, usually given in the form of input/output pairs. Lexicase outperformed other parent selection methods (e.g., tournament selection) on problems from the General Program Synthesis Program Suites (PSB1 and PSB2) [3, 6]. In a genetic algorithm setting to solve Boolean Constraint Satisfaction problems, lexicase outperformed fitness-proportionate and tournament selection [17]. In Automated Machine Learning (AutoML), lexicase converged to high-performing machine learning pipelines faster than Non-dominated Sorting Genetic Algorithm (NSGA-II) [16].

Various works have tried to explain lexicase's improved performance over other parent selection methods. For example, lexicase's ability to select and maintain 'specialists' is attributed as a key reason for its improved performance [5]. Specialists are individuals who have low errors on certain test cases with possibly a large overall total error. As such, specialists are less frequently selected by tournament selection and other error-aggregating parent selection methods. Another explanation for lexicase's performance is that it selects individuals residing on the corners of a high dimensional objective space [14]. Using diagnostics problems, [10, 11] found that lexicase is better at search space exploration than tournament selection. In terms of error diversity (number of unique error vectors in the population), lexicase seems

to maintain a high diversity compared to tournament selection in program synthesis problems [4].

Although lexicase selection does not have any parameters associated with the selection step, its performance depends on the underlying evolutionary parameter settings, such as population size and number of generations. In effect, we claim that these are some “hidden” parameters associated with lexicase selection. In [13], the authors study the effect of population size and number of test cases on the performance of lexicase selection for symbolic regression problems. Although the test set size affects the performance of lexicase, the effect is minimal for many problems, at least for the case when lexicase is applied in grammar-guided genetic programming setting [18]. In [20], the authors found that the success of lexicase in optimizing contradictory objectives depends on having a population size large enough relative to the number of objectives and generations.

19.3 Selection Scheme Diagnostics

We use the DOSSIER suite from [11] to analyze how population size affects the performance of lexicase selection under different search spaces and problem conditions. The suite provides a set of diagnostic tools to characterize a selection scheme’s exploitation and exploration abilities. Here, selection schemes and parent selection methods are synonymous. The DOSSIER suite comprises eight diagnostics, each using a unique search space with specific problem-solving characteristics (exploitation, exploration, and fitness-valley crossing). This suite has been used to illustrate differences between several lexicase variants [10, 20], understand the theoretical limits of lexicase [19], and phylogenetic work within evolutionary computation [8, 15, 21].

This work uses the exploitation rate diagnostic and the contradictory objectives diagnostic. Each diagnostic specifies a unique genotype to phenotype transformation; both the genotype and phenotype are numerical vectors of the same dimensionality. We refer to each value in a genotype as a gene and each value in a phenotype as a trait. Solutions consist of genotypes and are assigned phenotypes post-diagnostic evaluation. Selection schemes use the phenotype to identify parents (e.g., tournament selection can use an aggregated phenotype). The evolutionary framework outlined in [11] limits search space traversal to hill-climbing via only mutations. Doing so helps isolate and analyze the effect of lexicase as the primary driver of success or failure in a given search space.

19.3.1 *Exploitation Rate Diagnostic*

The exploitation rate diagnostic measures how fast a selection scheme can steer a population toward a single optimum. The optimum here refers to a phenotype

comprised of traits with the largest numerical value possible. Efficiently reaching an optimum is particularly crucial for problems with computationally expensive evaluation procedures. The search space for this diagnostic consists of a smooth, uphill gradient that leads directly toward the single optimum. To generate a solution's phenotype, each gene's value from a solution's genotype is directly copied into the corresponding trait position: the first value of the first gene is copied into the first trait, the value of the second gene is copied into the second trait, and so on. More specifically, phenotypes are copies of the given genotypes.

This diagnostic favors selection schemes that prioritize overall high-performing solutions, as parents exhibiting this characteristic are positioned closer to the optimum. Consequently, offspring from these parents can benefit from mutations pushing them toward the optimum. Truncation selection and tournament selection perform well on this diagnostic as they both aggregate all of a solution's traits and exhibit strong selection pressure for top-performing solutions [11]. Lexicase selection, however, has no explicit mechanism for selecting these parents. Instead, it selects specialists performing well on a subset of test cases, aiming to evolve solutions that eventually perform well on all cases (traits).

When working with a fixed evaluation budget, an evolutionary search must balance the number of generations and the population size. A larger population requires more parents to be identified per generation, which increases the chances of generating permutations of test cases to identify better-performing solutions. Moreover, a larger population will result in fewer generations when working with a fixed evaluation budget. Conversely, a smaller population increases the number of generations, which can lead to the accumulation of more mutations. Ultimately, this diagnostic allows us to measure how exploitation is affected by these two conflicting dynamics: (1) increasing population size to identify better-performing solutions and (2) increasing the number of generations to accumulate mutations.

19.3.2 *Contradictory Objective Diagnostic*

The contradictory objectives diagnostic captures how well a selection scheme can simultaneously pursue and reach multiple, identical optima within a population. The ability to reach multiple optima for a given problem is crucial for problems where no single solution exists and problems with conflicting objectives. The search space for this diagnostic consists of multiple identical, smooth, uphill gradients leading to unique optima. The diagnostic dimensionality determines the number of gradients and optima in the search space.

To generate a solution's phenotype, we scan through all the genes in the genotype, identify the gene with the greatest value (ties are broken by picking the gene closest to the beginning of the genotype), and label it as the activation gene. The value of the activation gene is then directly copied into the trait residing in the same position within the phenotype and all other traits are set to zero (Fig. 19.1). This genotype-to-phenotype transformation enforces that a given individual cannot work towards

		Active Gene				
		↔				
		g1	g2	g3	g4	g5
Genotype		57.4	26.8	64.6	77.2	67.1
		t1	t2	t3	t4	t5
Phenotype		0	0	0	77.2	0

Fig. 19.1 Example phenotype construction for the contradictory objectives diagnostic. Note that the trait values of all genes except for the maximum value are zero. In this case, all trait values serve as test cases

optimizing multiple traits simultaneously. As such, the complete set of optima consists of phenotypes, each with one trait set to the largest numerical value possible, while all other traits are set to zero.

This diagnostic favors selection schemes focused on finding Pareto-optimal solutions, as it generates a search space with multiple Pareto solutions. In fact, [11] found that nondominated sorting excels on this diagnostic. Selection schemes are challenged with maintaining a population consisting of solutions exploring multiple gradients while climbing these gradients to reach optima. Lexicase can effectively explore without sacrificing its ability to exploit, but its ability to explore and maintain specialists across different test cases is influenced by the population size [13]. When working with a fixed evaluation budget, the tradeoff between the population size and the number of generations will directly impact lexicase’s ability to maintain multiple gradients within the population. Moreover, the number of generations may impact the progress on a gradient.

19.3.3 Test Case Redundancy

To our knowledge, all current work using the DOSSIER suite from [11] paired with lexicase selection incorporate exactly one test case for each trait within the phenotype. This mapping from test case to trait creates an equal opportunity for each trait to appear first within the test case shuffling lexicase implements. Here, we present the test case redundancy diagnostic, which generates duplicate test cases to measure how an imbalanced representation of traits affects lexicase’s ability to maintain specialists. The test case redundancy diagnostic is more of an extension, like the fitness-valley crossing component within [11]. This diagnostic is particularly informative for lexicase selection, as prior work focuses on its ability to retain specialists for unique test cases [13].

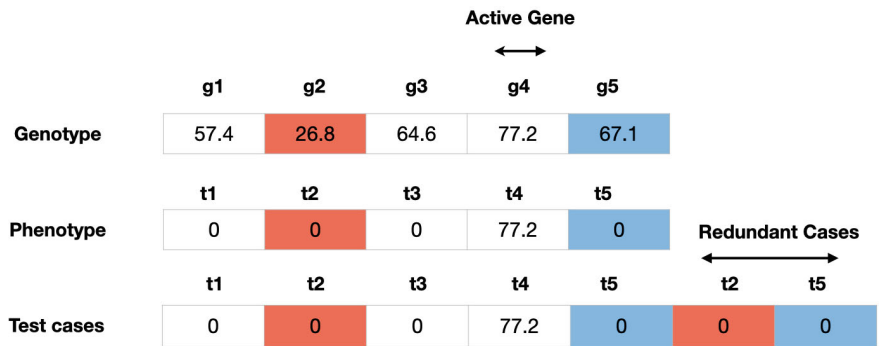


Fig. 19.2 Mapping from an example genotype to phenotype for test case redundancy diagnostic. The trait values corresponding to the colored genes are duplicated and placed at the end of the existing sequence of test cases

This diagnostic measures how many specialists lexicase can maintain within the population when redundant test cases are integrated within the contradictory objectives diagnostic. We extend the original set of test cases (the set of test cases in the standard contradictory objectives diagnostic) by appending redundant test cases. Each test case in the original set of test cases corresponds to a single trait within the phenotype, giving a total of 100 test cases. But here, multiple test cases can correspond to the same trait value giving us more than 100 test cases. Each additional test case is drawn from the original set of test cases with equal probability and replacement. The new test case set is constructed at the start of an evolutionary run and lexicase selection will use it for each selection event. See Fig. 19.2 for an example. In this work, we vary the number of additional test cases to 100, 200, and 400.

19.4 Methods

We investigated how the combination of population size, test case redundancy, and number of generations under a consistent evaluation budget influences lexicase selection’s ability to exploit gradients and maintain specialists. From the DOSSIER suite in [11], we used the exploitation rate diagnostic to measure exploitation and the contradictory objectives diagnostic to measure specialist maintenance. Additionally, we extended the contradictory objectives diagnostic by incorporating redundancy within the number of test cases associated with phenotypic traits. We conducted five sets of experiments: one with the exploitation rate diagnostic, one with the contradictory objectives diagnostic, and three with the test case redundancy diagnostic. For each experiment, we compared how varying population sizes (50, 100, 500, 1000, and 5000) impact performance with the given diagnostic. In all, we experimented with a combination of five different diagnostics and five different population sizes, giving

us a total of 25 treatments. We performed 50 replicates within each treatment and restricted runs to a 1.5×10^9 evaluation budget.

19.4.1 Evaluation Budget

In this work, we define the evaluation budget as the maximum number of times test cases are considered for parent selection throughout an evolutionary run. The population size and the number of test cases used by lexicase selection determine the number of evaluations for a single generation. For example, 100 test cases and 100 solutions will result in 10, 000 evaluations per generation. For a fixed evaluation budget and fixed number of test cases, the population size determines the number of generations allowed during an evolutionary search. We used an evaluation budget of 1.5×10^9 evaluations. As such, we adjusted the number of generations allowed for each population size used in this work. Table 19.1 demonstrates the number of generations for each lexicase configuration when evaluated on a specific diagnostic.

19.4.2 Diagnostic Experiments

We set the dimensionality to 100 for the diagnostics used in this work, resulting in 100 genes and traits. However, the number of total test cases used by lexicase selection varies by the diagnostic used. For experiments with the exploitation rate diagnostic (Sect. 19.3.1) and the contradictory objectives diagnostic (Sect. 19.3.2), the standard 100 test cases are used to identify parent solutions (as in [11]). For experiments with the test case redundancy diagnostic (Sect. 19.3.3), the total number

Table 19.1 Number of generations lexicase selection with a specific population size and number of test cases can execute for a 1.5×10^9 evaluation budget. Note that the standard version of the exploitation rate diagnostic and the contradictory objectives diagnostic are being considered when the redundancy of cases is set to 0. For redundancy of test cases greater than 0, the number of redundant cases is added within the contradictory objectives diagnostic

		Redundancy of cases			
		0	100	200	400
Population size	50	300,000	150,000	100,000	60,000
	100	150,000	75,000	50,000	30,000
	500	30,000	15,000	10,000	6,000
	1000	15,000	7,500	5,000	3,000
	5000	3,000	1,500	1,000	600
Size of test case set		100	200	300	500

of test cases varies by the number of redundant test cases added (100 sampled test cases generates 200 total test cases, 200 sampled test cases generates 300 total test cases, etc.). The bottom row of Table 19.1 summarizes the total number of test cases for each redundancy count.

At the start of an evolutionary run, the initial population is comprised of solutions with genotypes consisting of random values between 0.0 and 1.0. This process is used to start with a set of random solutions that reside near the lowest point in the search space. Once the initial population is constructed, each solution is evaluated with the given diagnostic and assigned a phenotype. Lexicase then uses the set of test cases to identify parents where the total number of identified parents is equal to the size of the initial population. The number of solutions within the initial population depends on the population size configuration of the particular experiment.

Each identified parent reproduces asexually and there is a 0.07% chance of a point mutation being applied to each gene within an offspring's genotype. The magnitude of a point mutation is drawn from a Gaussian Distribution with the mean set to 0.0 and the standard deviation set to 1.0. If a mutation results in a negative gene, we set the gene to its absolute value (e.g., a gene of -0.7 post-mutation would be adjusted to 0.7). If a mutation causes a gene to exceed the upper threshold (100.0), we rebound it by calculating the excess over the upper threshold and subtracting this excess from the upper threshold (e.g., a gene of 102.3 post-mutation would be adjusted to 97.7). This rebounding process was used to evaluate lexicase's ability to refine solutions near optima. The set of offspring make up the following population and the same cycle is repeated; the number of generations depends on the particular experiment configuration (Table 19.1).

19.4.3 Data Tracking

The exploitation rate diagnostic measures how fast a selection scheme can hill-climb. We collect performance data to determine how close a population has come to the optimum, and we use the total number of evaluations accumulated each generation to define time. We calculated the average trait score (performance) for each solution's phenotype by summing all the traits and dividing the sum by the dimensionality (100); performance falls between 0.0 and 100.0. We record the best performance found in the population each generation. Additionally, we record the total number of evaluations accumulated when a satisfactory solution is found. A solution is deemed satisfactory if the phenotype consists of all satisfactory traits, where a satisfactory trait is defined as a trait greater than or equal to 99.0. Recording when a satisfactory solution is found illustrates how efficiently lexicase can reach them.

All other diagnostics in this work focus on specialist maintenance under various conditions. We collect activation genes from all solutions in the population and count the number of unique activation genes to calculate the activation gene coverage. This coverage value demonstrates lexicase's ability to maintain multiple specialists within the population, although they may not be satisfactory. We collect all of the

satisfactory traits found in the population and count the number of unique satisfactory traits to calculate satisfactory trait coverage. This coverage value illustrates lexicase's ability to simultaneously push multiple specialists toward becoming satisfactory. Activation gene coverage ranges between 1 and 100 and satisfactory trait coverage ranges between 0 and 100. Note that redundant test cases do not alter these coverage ranges, as redundant test cases do not create new genes or traits.

19.4.4 Statistical Analysis

We conducted a Kruskal-Wallis test to determine if significant differences occurred between the population sizes. If comparisons resulted in significant differences with the Kruskal-Wallis test, we performed a post-hoc Wilcoxon rank-sum test to identify differences among population size pairings with a Bonferroni correction for multiple comparisons. We used a significance level of 0.05 for all statistical tests.

19.4.5 Software Availability

Our supplemental material [12] is hosted on GitHub and contains all files related to software, data analysis, figure visualization, and documentation for this work. We used Python to implement our experiments and R version 4 for statistical testing and data visualizations. All data is available on the Open Science Framework at <https://osf.io/g5u9p/>.

19.5 Results and Discussion

19.5.1 Smaller Population Sizes Facilitate Faster Exploitation

We used the exploitation rate diagnostic to measure how population size affects lexicase selection's ability to hill-climb given a fixed evaluation budget. Figure 19.3 illustrates the results for lexicase with varying population sizes on this diagnostic. For the population sizes used in this work, only sizes 50, 100, and 500 evolved satisfactory solutions within the given computational budget (Fig. 19.3c). We found that a population size of 500 required more evaluations to find satisfactory solutions than both sizes of 100 and 50, and the population size of 50 found satisfactory solutions faster than a size of 100 (Wilcoxon rank-sum test: $p < 10^{-4}$).

Each configuration of lexicase improved performance over time (Fig. 19.3a), albeit improving at different rates. We can see that smaller population sizes exhibit faster improvement, with only sizes 50, 100, and 500 reaching the optimum by the end of

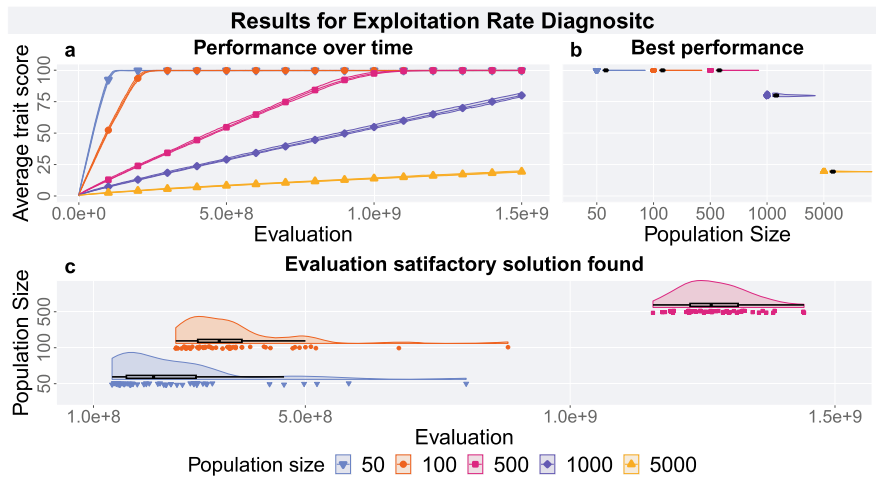


Fig. 19.3 Results for lexicase selection with varying population sizes on the exploitation rate diagnostic. We report **a** the best performance in the population over time, **b** the best performance evolved throughout the evolutionary run, and **c** the total accumulated evaluations when a satisfactory solution was first discovered. For panel **a**, we plot the average performance with surrounding boundaries from the best and worst performances across the 50 replicates for every 10^8 evaluations

an evolutionary run. The trajectory of improvement for population sizes of 1000 and 5000 indicates that they can reach optimum given a larger evaluation budget. Figure 19.3b reports the best performance evolved throughout an evolutionary search. Interestingly, we detected no difference when comparing these performances between sizes 50 and 100 (Wilcoxon rank-sum test: $p > 0.05$), and both of these sizes evolved better performances than all other sizes (Wilcoxon rank-sum test: $p < 10^{-3}$). Among the remaining population sizes, the size of 5000 resulted in the worst performances, while the size of 1000 produced worse solutions than the size of 500 (Wilcoxon rank-sum test: $p < 10^{-4}$).

These results provide evidence that smaller population sizes facilitate more hill-climbing given a fixed evaluation budget. Beneficial and harmful mutations have an equal probability of occurring. Whereas solutions that accumulate beneficial mutations improve overall performance and the likelihood of becoming parents, the solutions accumulating harmful mutations will likely not be identified as parents. Given a fixed evaluation budget, smaller population sizes get more generations to evolve solutions, fewer selection events per generation, and more opportunities to accumulate beneficial mutations. Larger population sizes, however, get fewer generations to evolve solutions and more selection events per generation, but at the expense of limited mutations accumulating over time.

This diagnostic allows us to intuitively understand how the tradeoff between smaller and larger population sizes affects lexicase’s ability to exploit. To our knowledge, this is the first work looking directly at lexicase’s ability to exploit and the

selection pressure it exhibits toward top-performing solutions with varying population sizes. Interestingly, [2] hypothesized that the downsampled variations of lexicase benefited from deeper evolutionary searches. Here, we find additional evidence illustrating the advantages of using a smaller per-generation evaluation cost (i.e. a smaller population size or a smaller set of test cases).

19.5.2 Larger Population Sizes Yield More Optimal Specialists When No Redundancy is Added

We used the contradictory objectives diagnostic to measure how population size affects lexicase selection's ability to maintain and optimize specialists when there is no redundancy of test cases. Figure 19.4 illustrates the results for lexicase on this diagnostic. At the start of an evolutionary run, there is nearly perfect activation gene coverage for population sizes of 500, 1000, and 5000, while the remaining sizes begin with low coverage (Fig. 19.4b). High activation coverage is expected with large sizes as the initial population consists of randomly generated genotypes with genes ranging between 0.0 and 1.0. Interestingly, the population size of 5000 maintains perfect coverage throughout the entire evolutionary run for all replicates. The remaining sizes, however, see a decline in activation gene coverage and converge to different coverage values at the end of an evolutionary run.

Activation gene coverage illustrates how many specialists lexicase has maintained in the population, while satisfactory trait coverage demonstrates how many of those specialists reach satisfactory levels. Figure 19.4a demonstrates the growth in satisfactory trait coverage over time. While the smaller sizes reached satisfactory traits quickly, they did not reach high coverage like larger sizes. Panel (c) in Fig. 19.4 reports the best satisfactory trait coverage throughout an evolutionary run. Interestingly, the population size of 5000 evolves populations with perfect satisfactory trait coverage (100) for each replicate. All other population sizes reached lower coverage, where larger sizes reached better satisfactory trait coverage than smaller sizes (Wilcoxon rank-sum test: $p < 10^{-2}$).

Overall, our results are consistent with previous work illustrating that lexicase can maintain more specialists with larger populations [10, 13]. Interestingly, all possible specialists were optimized to satisfactory levels with a population size of 5000. These results provide additional evidence that illustrates lexicase's ability to maintain specialists is determined by the population size.

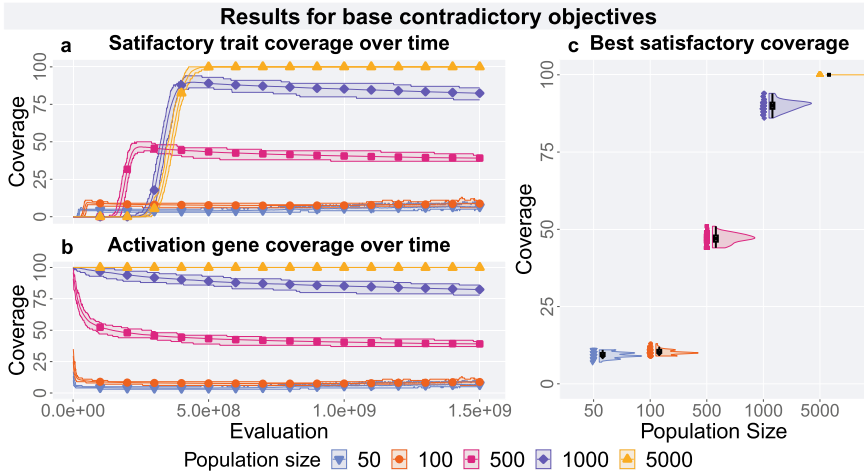


Fig. 19.4 Results for lexicase selection with varying population sizes on the contradictory objectives diagnostic. We report **a** the satisfactory trait coverage and **b** the activation gene coverage in the population over time, and **c** the best satisfactory trait coverage found throughout an evolutionary run. For panels **a** and **b**, we plot the average average with surrounding boundaries from the best and worst coverage across the 50 replicates for every 10^8 evaluations

19.5.3 Intermediate Levels of Redundancy Restrict Specialist Optimality with Large Population Sizes

Incorporating redundancy through duplicate test cases alters the selection dynamic of lexicase selection. In this section, we discuss the results of adding 100 and 200 redundant test cases to the set of test cases used by the lexicase. The results for 100 redundant cases are shown in Fig. 19.5, while the results for 200 redundant cases are displayed in Fig. 19.6.

Interestingly, adding either 100 or 200 redundant test cases produced similar results. In both cases, larger population sizes start with high activation gene coverage. However, coverage falls for most sizes within a few generations. The population size of 5000 maintained high coverage throughout the entire evolutionary run. In contrast, sizes 1000 and 500 maintain moderate coverage, while sizes 50 and 100 maintain low coverage. Regarding satisfactory trait coverage, the population size of 5000 maintained the most coverage and continuously improved coverage over time. For sizes 1000 and 500, there is an increase in satisfactory trait coverage over time, but the coverage eventually plateaus. The population sizes 50 and 100 have poor coverage throughout the evolutionary run. At the end of the respective runs, the population size of 5000 produced the populations with the best satisfactory coverage, followed by larger sizes producing more coverage than smaller sizes (Wilcoxon rank-sum test for all pairs; $p < 10^{-4}$).

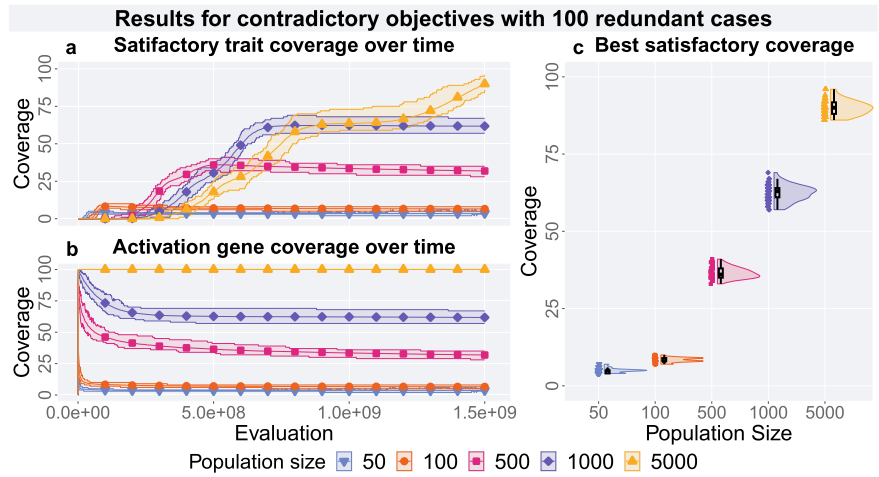


Fig. 19.5 Results for contradictory objectives with 100 redundant test cases We report **a** the satisfactory trait coverage and **b** the activation gene coverage in the population over time, and **c** the best satisfactory trait coverage found throughout an evolutionary run. For panels **a** and **b**, we plot the average average with surrounding boundaries from the best and worst coverage across the 50 replicates for every 10^8 evaluations

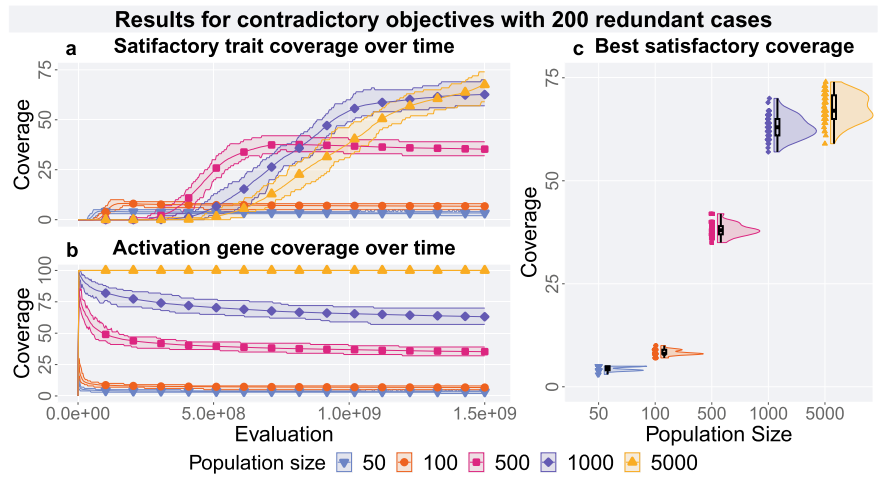


Fig. 19.6 Results for contradictory objectives with 200 redundant test cases We report **a** the satisfactory trait coverage and **b** the activation gene coverage in the population over time, and **c** the best satisfactory trait coverage found throughout an evolutionary run. For panels **a** and **b**, we plot the average average with surrounding boundaries from the best and worst coverage across the 50 replicates for every 10^8 evaluations

Looking at Fig. 19.5c (redundancy of 100) and Fig. 19.6c (redundancy of 200), one clear trend emerges. The best satisfactory coverage decreases with an increase in

redundancy. One explanation for this behavior is that with redundancy in test cases, the probability of non-duplicate test cases appearing at the beginning of the shuffle in lexicase decreases. As a result, larger population sizes are needed to increase the chances of all test cases appearing first in the shuffle. Let us take an example to illustrate this point. Consider this sequence of test cases: a, b, c, d . Without redundancy, every test case has an equal probability of coming first in the shuffle. Consider the same sequence with redundancy added: a, a, b, b, c, d . Clearly, a and b are more probable to come first in the shuffle than c and d . Therefore, lexicase selection is more likely to maintain specialists for a, b .

19.5.4 Intermediate Population Sizes Produce a Greater Number of Optimal Specialists with Numerous Redundant Test Cases

Figure 19.7 illustrates how lexicase selection performed on the test case redundancy diagnostic with our most extreme level of redundancy—400 redundant cases. Interestingly, the population size of 5000 excelled with all smaller levels of redundancy, reaching higher satisfactory trait coverage than all other sizes (Wilcoxon rank-sum test: $p < 10^{-4}$). With the 400 redundant test cases, however, the population size of 5000 does not achieve the greatest satisfactory trait coverage.

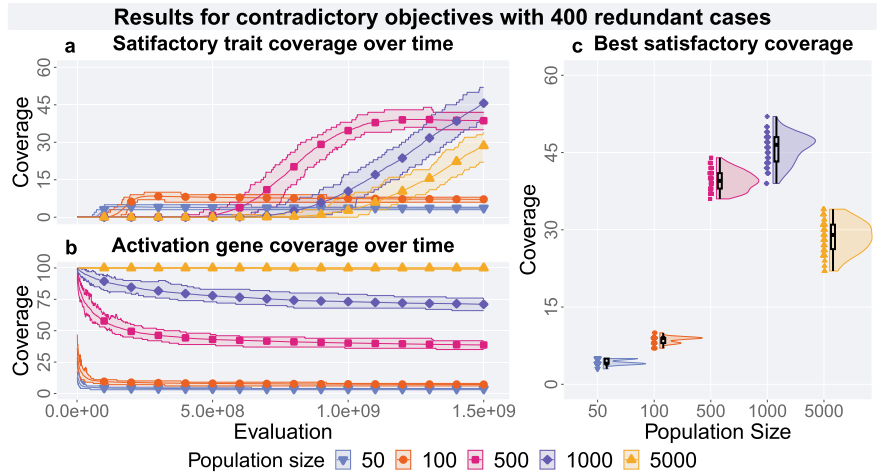


Fig. 19.7 Results for lexicase selection with varying population sizes on the contradictory objectives diagnostic with 400 redundant test cases. We report **a** the satisfactory trait coverage and **b** the activation gene coverage in the population over time, and **c** the best satisfactory trait coverage found throughout an evolutionary run. For panels **a** and **b**, we plot the average average with surrounding boundaries from the best and worst coverage across the 50 replicates for every 10^8 evaluations

The activation gene coverage maintained by the different population sizes follows a similar trend to the previous results (Fig. 19.7b). The population size of 5000 maintained nearly perfect activation gene coverage throughout the entire run for all replicates. The remaining population sizes maintained less coverage, where larger sizes maintained more coverage than smaller sizes. While the size of 5000 managed to maintain nearly perfect activation gene coverage, it was unable to reach high satisfactory trait coverage like in the previous results. In fact, the population size of 5000 reached lower satisfactory trait coverage than both sizes 1000 and 500, and the size of 1000 reached greater coverage than the size of 500 (Wilcoxon rank-sum test: $p < 10^{-4}$). The remaining population sizes maintained less coverage, with size 100 generating more coverage than size 50 (Wilcoxon rank-sum test: $p < 10^{-4}$).

Figure 19.7a illustrates the growth in satisfactory trait coverage over time for the different population sizes. Interestingly, the population sizes of 5000 and 1000 appear to improve satisfactory trait coverage throughout the evolutionary run, whereas all other sizes plateau. The high activation gene coverage maintained by the size 5000 (19.7b) and the growth trajectory in satisfactory trait coverage (19.7a) suggest that it might eventually reach perfect satisfactory trait coverage given a larger evaluation budget, surpassing all other sizes. The per-generation evaluation cost is now a crucial aspect to consider given that 500 test cases must now be considered. For larger population sizes, the per-generation evaluation cost is 250, 000 for a size of 500, 500, 000 for a size of 1000, and 2, 500, 000 for a size of 5000. These results present a clear tradeoff between using different population sizes for maintenance and optimizing specialists with an evaluation budget.

19.6 Conclusion

In this work, we used the DOSSIER suite in [11] to examine how lexicase selection's capacity for exploitation and specialist maintenance is affected by different population sizes given a consistent evaluation budget. We used the exploitation rate diagnostic to capture lexicase's ability to exploit and the contradictory objectives diagnostic to capture lexicase's ability to maintain specialists. Additionally, we extended the contradictory objectives diagnostic by incorporating redundant test cases into the pool of test cases lexicase shuffles to consider a new dynamic regarding specialist maintenance.

Lexicase selection is typically presented as being parameter-less when compared against other selection schemes, as all the parameters used by an evolutionary algorithm are predefined (e.g. population size, variation operators, mutation rate, etc.). However, when comparing lexicase to itself with a fixed computational budget and a given problem, it is clear that the population size is a varying parameter that can affect its performance. All practitioners implicitly use an evaluation budget (the number of times test cases are considered during parent selection) when using lexicase selection, whether they are aware of it or not. As such, the question becomes what is

the ideal parameter setting for lexicase selection given a problem being solved and an evaluation budget?

Our results illustrate that there is no perfect population size for all problems and the characteristics of a given problem must be considered for the best configuration of the evolutionary algorithm paired with lexicase selection. The results of the exploitation rate diagnostic present evidence that smaller population sizes should be preferred for problems that require exploitation to reach optima. To our knowledge, this is the first work analyzing how population size affects lexicase selection's ability to exploit and steer a population toward optima. Conversely, the results of the contradictory objectives diagnostic present evidence that larger population sizes should be preferred for problems that require specialists to be maintained to reach optima. This result is consistent with previous work illustrating that lexicase's ability to maintain specialists is determined by the population size being used.

We considered a new aspect regarding lexicase selection's ability to maintain specialists. Previous works often assume that a unique test case must come first in the shuffle to allow a specialist to be selected. Here, we examine the scenario where duplicate test cases exist for a given trait within the contradictory objectives diagnostic. The addition of duplicate test cases creates a new dynamic as the probability of a specialist for a particular test case being selected now varies and the per-generation evaluation cost is increased. We found that with 100 and 200 redundant test cases, larger population sizes can maintain more specialists, with the largest population size (5000) used in this study performing the best. However, we found the largest population size was outperformed by smaller sizes (1000 and 500) when 400 redundant test cases were used. These results highlight the importance of considering how many test cases to use for a given problem and the information each test case captures to guide the evolutionary search. Clearly, having correlated test cases may lead to an excessive number of test cases to be considered, potentially wasting evaluations when working within a constrained budget.

Ultimately, these diagnostics enable us to dissect lexicase selection under various conditions and provide an intuitive understanding of why each condition yields a particular effect. For future work, it would be interesting to compare how more complex variants of lexicase selection differ from one another (e.g., downsampled lexicase).

Acknowledgements We thank members of the Department of Computational Biomedicine at Cedars-Sinai Medical Center for their helpful comments on this work. Additionally, we thank Ting Hu and Lee Spector for their suggestions during the review process. Cedars-Sinai Medical Center provided computational resources through their High Performance Computing clusters. The work was supported by NIH grants R01 LM010098 and U01 AG066833.

References

1. Boldi, R., Briesch, M., Sobania, D., Lalejini, A., Helmuth, T., Rothlauf, F., Ofria, C., Spector, L.: Informed down-sampled lexibase selection: identifying productive training cases for efficient problem solving. In: *Evolutionary Computation*, pp. 1–32 (2024)
2. Ferguson, A.J., Hernandez, J.G., Junghans, D., Lalejini, A., Dolson, E., Ofria, C.: Characterizing the effects of random subsampling on lexibase selection. In: Banzhaf, W., Goodman, E., Sheneman, L., Trujillo, L., Worzel, B. (eds.) *Genetic Programming Theory and Practice XVII*, pp. 1–23. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-39958-0_1
3. Helmuth, T., Abdelhady, A.: Benchmarking parent selection for program synthesis by genetic programming. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pp. 237–238 (2020)
4. Helmuth, T., McPhee, N.F., Spector, L.: Effects of lexibase and tournament selection on diversity recovery and maintenance. In: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pp. 983–990 (2016)
5. Helmuth, T., Pantridge, E., Spector, L.: On the importance of specialists for lexibase selection. *Genet. Program. Evolvable Mach.* **21**(3), 349–373 (2020)
6. Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference—GECCO '15*, pp. 1039–1046. ACM Press, Madrid, Spain (2015). <https://doi.org/10.1145/2739480.2754769>. <http://dl.acm.org/citation.cfm?doid=2739480.2754769>
7. Helmuth, T., Spector, L., Matheson, J.: Solving uncompromising problems with lexibase selection. *IEEE Trans. Evol. Comput.* **19**(5), 630–643 (2014)
8. Hernandez, J.G., Lalejini, A., Dolson, E.: What can phylogenetic metrics tell us about useful diversity in evolutionary algorithms? In: Banzhaf, W., Trujillo, L., Winkler, S., Worzel, B. (eds.) *Genetic Programming Theory and Practice XVIII*, pp. 63–82. Springer Nature Singapore, Singapore (2022). https://doi.org/10.1007/978-981-16-8113-4_4
9. Hernandez, J.G., Lalejini, A., Dolson, E., Ofria, C.: Random subsampling improves performance in lexibase selection. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 2028–2031 (2019)
10. Hernandez, J.G., Lalejini, A., Ofria, C.: An exploration of exploration: Measuring the ability of lexibase selection to find obscure pathways to optimality. In: Banzhaf, W., Trujillo, L., Winkler, S., Worzel, B. (eds.) *Genetic Programming Theory and Practice XVIII*, pp. 83–107. Springer Nature Singapore, Singapore (2022). https://doi.org/10.1007/978-981-16-8113-4_5
11. Hernandez, J.G., Lalejini, A., Ofria, C.: A suite of diagnostic metrics for characterizing selection schemes (2023)
12. Hernandez, J.G., Saini, A.K., Moore, J.H.: Supplemental Material GitHub Repository (2024). <https://doi.org/10.5281/zenodo.11199991>. <https://github.com/jgh9094/GPTP-2024-Lexibase-Analysis/>
13. La Cava, W., Helmuth, T., Spector, L., Moore, J.H.: A probabilistic and multi-objective analysis of lexibase selection and ϵ -lexibase selection. *Evol. Comput.* **27**(3), 377–402 (2019)
14. La Cava, W., Spector, L., Danai, K.: Epsilon-lexibase selection for regression. In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pp. 741–748 (2016)
15. Lalejini, A., Moreno, M.A., Hernandez, J.G., Dolson, E.: Phylogeny-informed fitness estimation for test-based parent selection. In: Winkler, S., Trujillo, L., Ofria, C., Hu, T. (eds.) *Genetic Programming Theory and Practice XX*, pp. 241–261. Springer Nature Singapore, Singapore (2024). https://doi.org/10.1007/978-981-99-8413-8_13
16. Matsumoto, N., Saini, A.K., Ribeiro, P., Choi, H., Orlenko, A., Lyytikäinen, L.P., Laurikka, J.O., Lehtimäki, T., Batista, S., Moore, J.H.: Faster convergence with lexibase selection in tree-based automated machine learning. In: *European Conference on Genetic Programming (Part of EvoStar)*, pp. 165–181. Springer (2023)
17. Metevier, B., Saini, A.K., Spector, L.: Lexibase Selection Beyond Genetic Programming, pp. 123–136. Springer (2019)

18. Schweim, D., Sobania, D., Rothlauf, F.: Effects of the training set size: a comparison of standard and down-sampled lexicase selection in program synthesis. In: 2022 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8. IEEE (2022)
19. Shahbandegan, S., Dolson, E.: Theoretical limits on the success of lexicase selection under contradictory objectives. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation, GECCO '23 Companion, pp. 827–830. Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3583133.3590714>
20. Shahbandegan, S., Dolson, E.: On the robustness of lexicase selection to contradictory objectives. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '24, pp. 594–602. Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3638529.3654215>
21. Shahbandegan, S., Hernandez, J.G., Lalejini, A., Dolson, E.: Untangling phylogenetic diversity's role in evolutionary computation using a suite of diagnostic fitness landscapes. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '22, pp. 2322–2325. Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3520304.3534028>

Chapter 20

Using Lineage Age to Augment Search Space Exploration in Lexicase Selection



Karen Suzue, Charles Ofria, and Alexander Lalejini

Abstract Evolutionary algorithms often become stuck on a particular evolutionary trajectory, limiting the available outcomes and often excluding global optima. Indeed, the starting state of a population (or early decisions) may limit the regions of search space that the algorithm will ultimately explore. One mechanism used to ensure that more regions of a search space are considered is to regularly inject new random starting points, while giving special advantages to younger lineages to give them a chance to survive long enough to realize their potential. In this chapter, we explore including periodic injections of random solutions into lexicase selection, along with age-based selection criteria. We demonstrate this technique's potential for increased exploration using both program synthesis benchmark problems and construction of Sudoku boards with particular solving characteristics. Our results are promising, but inconsistent, ranging from highly effective to no meaningful difference from controls. Ultimately, we provide directions for future research that might more fully realize the power of age-based criteria in lexicase selection.

20.1 Introduction

Premature convergence remains a longstanding obstacle for evolutionary algorithms, hindering their ability to produce optimal solutions. When a population converges to a local optimum and lacks meaningful genetic diversity, it can become stuck in that region of the search space. While numerous strategies have been devised to combat premature convergence, the predominant paradigm centers on diversity maintenance.

K. Suzue (✉) · C. Ofria

Department of Computer Science and Engineering and Program in Ecology, Evolution and Behavior,
Michigan State University, East Lansing, MI, USA

e-mail: suzuekar@msu.edu

C. Ofria

e-mail: ofria@msu.edu

A. Lalejini

School of Computing, Grand Valley State University, Allendale, MI, USA

e-mail: lalejina@gvsu.edu

This approach involves preserving internally discovered genetic material to prevent the population from losing valuable building blocks or inferior prospective solutions that remaining promising in the longer term.

While diversity maintenance is known for its efficacy in safeguarding existing diversity, another effective approach for improving search space exploration involves the external introduction of novel genetic material into the population. This alternative paradigm addresses two shortcomings of diversity maintenance. First, injecting new genetic material allows the algorithm to explore solutions beyond the attractor basins determined by the initial population [18]. Second, a genetic inflow eliminates dependence on the pre-existence of useful diversity within the population, which is not always guaranteed. However, a key challenge with this alternative approach is ensuring the preservation of introduced genetic material, as brand new prospective solutions will likely start off unfit and thus fail to be selected for propagation. Conventional methods like Age-Layered Population Structure (ALPS) [18–20] and Hierarchical Fair Competition (HFC) [21] address this challenge by explicitly structuring the population to preserve new lineages, sustaining their presence for further exploration and potential selection.

In this work, we explore mechanisms to augment lexica selection by continuously injecting fresh genetic material. Lexica selection is designed for test-based problems where different aspects of a candidate solution's quality are assessed by measuring its performance across a set of test cases [14]. Lexica selection has been shown to be successful across a wide range of domains [1, 6, 25, 26, 30–32], and lexica's success is often attributed to its efficacy in maintaining diversity [7, 12]. Unlike traditional selection schemes that evaluate individuals based on a single aggregated fitness score, lexica selection adopts a granular approach. Lexica selection requires a breakdown of fitness into distinct objectives or test cases, evaluating individuals' performances across these criteria independently. This method of fitness assessment enables the coexistence of subpopulations across diverse regions of the search space, with each group focusing on a specific aspect of the problem at hand. As a result, lexica selection is able to thoroughly and simultaneously explore many different pathways [15, 16], thereby mitigating premature convergence.

Despite its effectiveness, lexica selection still encounters challenges related to search space exploration. Specifically, its ability to fully explore the search space depends on the ratio between the number of test cases and population size, determining whether it can ensure the presence of specialists across all subsets of criteria [7, 15, 17]. As expected, this task becomes particularly challenging with a small population size relative to the number of test cases. Moreover, considering lexica selection's use of elite selection for each of its test cases, individual lineages can still become trapped in local optima. Indeed, these observations align with the findings of a prior study by Hernandez et al. [16] that identified lexica's poor capacity for crossing fitness valleys.

By supplying lexica selection with a continuous stream of external genetic material, we hypothesize that exploration will be further improved. To ensure the survival of newly injected individuals, we draw inspiration from ALPS and incorporate an age-tracking system. Leveraging lexica's use of test cases, we add a test

case selecting for the lineage that was injected into the population most recently (i.e., the youngest “age”). This approach prolongs the survival of newly injected individuals, enhancing their competitiveness against older, more established genotypes in the population, while still maintaining the most successful candidate solutions for various combinations of test cases. Furthermore, this strategy eliminates the need to directly manage population structure, as required in ALPS. In essence, age-based lexicase not only addresses the shortcomings of both lexicase and ALPS but also capitalizes on their effective diversity maintenance and exploration mechanisms.

We evaluated age-based lexicase selection on five program synthesis benchmark problems. Age-based lexicase improved problem-solving success for one of the five problems. While four program synthesis problems showed no improvement when selecting for age, in no case were the results significantly worse. We further demonstrate that this technique can also help with other, unrelated problem types, such as constructing Sudoku puzzles that exhibit targeted solving properties.

20.2 Background

The simplest strategies to mitigate premature convergence involve delaying its occurrence. Such mitigation can be achieved by adjusting mutation rate, mutation size, and population size, all of which are superficially simple [18]. However, many issues arise when using these straightforward approaches. While high mutation rates generate diversity, they do so at the cost of destroying existing building blocks [4, 5, 8, 18]. Moreover, this strategy fails to ensure the survival of new genotypes, which struggle to compete against older and well-adapted individuals in the population. Configuring a large mutation size creates significant variation between offspring and parents, however, this can hinder the population’s exploration of narrow pathways in the fitness landscape [18]. Determining an appropriate population size is difficult, and for some problems, the necessary size may exceed practical feasibility [18].

More sophisticated strategies, such as diversity maintenance techniques, focus on preserving existing genetic material within the population by introducing new mechanisms rather than simply relying upon parameter adjustment. Many of these techniques emulate ecological niches, allowing a diverse array of subpopulations to coexist [23, 29, 33, 35]. One such technique is crowding, which is implemented during the replacement stage of evolution, where individuals from the current pool and their offspring are selected for the next generation. Crowding involves comparing offspring with similar individuals in the current population to determine which genotype will persist [9]. This technique effectively prevents individuals from evolving in densely populated regions of the search space, thereby mimicking competition for resources in nature. Similarly, fitness sharing decreases the likelihood of selection based on individuals’ genotypic similarities, but does so implicitly by penalizing their fitness [37]. Island models impose additional structure onto the population, evolving subpopulations independently with limited migration occurring between them [2].

Premature convergence can also be mitigated by introducing external genetic material. Conventional techniques within this paradigm employ various mechanisms to ensure the survival of novel genetic material, thus facilitating sufficient exploration to offset exploitation. Hierarchical Fair Competition (HFC) is one such technique, which builds on the understanding that competition in many systems, including natural evolution, occurs at different levels [21]. For example, instead of direct competition between small and large organisms, species typically compete with others of similar sizes.

In HFC, the population is structured into layers based on fitness values, with each layer accommodating individuals whose fitness falls within a specified range. Migration in HFC occurs solely from lower to higher fitness layers to replace the worst genotypes within the latter. In theory, these mechanisms allow HFC to continuously pass promising building blocks from preceding layers onto subsequent layers, facilitating a synthesis of increasingly refined solutions. This algorithm requires several parameters to be determined, including the number of layers, the number of subpopulations per layer, and the admission and export thresholds for each layer. Results from the original study on the even n -parity benchmark demonstrate that HFC variants consistently outperform both single and multi-population (island model) genetic programming (GP) across a range of population sizes. Furthermore, with the introduction of novel genetic material, HFC is robust at small population sizes, thereby reducing the necessity for large populations [21].

HFC implicitly promotes the survival of the youngest, newly injected individuals, assuming they often have the lowest fitness. Age-Layered Population Structure (ALPS), however, implements this concept explicitly. Similar to HFC, ALPS stratifies the population according to age-based layers, with the number of layers and age intervals for each layer configured by the user. Here, age signifies how long genetic material has evolved within the population, not just an individual's generational count. Individuals accumulate age by 1 for every generation they persist. Injected newcomers have an age of 0, while offspring from mutation or recombination inherit an age of 1 plus their oldest parent's age [18].

In ALPS, new subpopulations are randomly generated in the first layer at regular predetermined intervals, replacing the previous individuals. Selection, breeding, and replacement exclusively occur between adjacent layers. These mechanisms allow individuals to compete with those of similar ages, enabling younger individuals to propagate their genes upward through the hierarchy. When organisms become too old for their layer, they are compared against the least fit individual in the subsequent layer. If their fitness surpasses that of the latter, they replace them. Consequently, individuals are gradually replaced unless they have reached global optima [18].

ALPS has been tested across several benchmark suites and applied to a diverse array of domains [42, 43]. The original study shows ALPS significantly outperforming both canonical and multistart EAs, as well as an adaptive variant of HFC, in an antenna optimization problem [18]. Interestingly, adaptive HFC shows no statistically significant improvement over the canonical EA on this specific problem, suggesting that while introducing novel genetic material and restricting competition may enhance exploration, using lineage age over fitness yields better results. In

another study, steady-state ALPS (ALPS-SS) is compared with CMAS-DE, DE, and simple GA on a similar antenna optimization problem, as well as unimodal and multimodal benchmark suites [19]. ALPS-SS consistently excels in multimodal problems, outperforming other algorithms while sustaining no loss of genetic diversity throughout a significantly large number of evaluations. However, compared to CMAS-DE and DE, ALPS-SS tends to underperform in unimodal problems, although it still outperforms simple GA.

Beyond ALPS and HFC, other age-based strategies have also demonstrated effectiveness in achieving higher quality solutions and better exploratory capabilities. Prior to ALPS and HFC, notable methods include Huber and Mlynski's Age-Controlled EA [22], as well as Kubota and Fukuda's genetic algorithm with age structure (ASGA) [24]. Concurrent with the development of ALPS and HFS are Conceicao Antonio's Hierarchical Genetic Algorithm (HGA) [3] and Schmidt and Lipson's Age-Fitness Pareto Optimization (AFP) [38]. AFP shares similarities with our work, as age is handled as a secondary objective in a multi-objective problem. On a set of randomly generated symbolic regression problems, AFP outperforms ALPS by discovering the target solution more frequently and with less computational effort [38]. More recent approaches include Real et al.'s AmoebaNetA [36] and Pozzuoli et al.'s Age-Layered MAP-Elites [34]. Age-Layered MAP-Elites similarly combines ALPS with MAP-Elites, another algorithm that emphasizes diversity and quality within a population of solutions.

We are aware of two existing methods of incorporating age-based selection into lexicase selection: ALPS lexicase [10] and Age-Mediated Parent Selection [40]. ALPS lexicase uses the explicit age-based population structure of the original ALPS algorithm but uses lexicase selection to mediate competition within each layer. Recent benchmarking experiments [10] indicate that ALPS-integrated lexicase performs better than most selection methods when tested on a programming synthesis benchmark suite, positioning it as the third-best performer behind down-sampled and MAD-CAP epsilon lexicase. Age-mediated parent selection (AMPS), as described in [40], is implemented to provide an additional diversification technique that can operate independent of lexicase selection. To our knowledge, the AMPS approach has yet to be rigorously benchmarked in the context of lexicase.

20.3 Methods

We assessed age-based lexicase selection on five program synthesis benchmark problems [11, 13] and constructing Sudoku puzzles with configurable solving experiences. For all problems, we compared the performance of random injection with age-based selection to two controls: random injection without age-based selection, and standard lexicase with no random injection or age-based selection.

20.3.1 *Lexicase Selection*

The lexicase parent selection algorithm is designed for test-based problems wherein candidate solutions are assessed using a set of training cases that specify correct behavior [39]. To select a parent, lexicase selection first shuffles the order of training cases, and all members of the population are added to a pool of candidates eligible for selection. Lexicase selection applies each training case in shuffled order to sequentially filter the pool of available candidates. For each training case, all but the elite candidates (on the current training case) are removed from the eligible pool. This process continues until all training cases have been used, and if more than one candidate remains in the pool, one is selected at random. The process starts over from the beginning, once again considering all of the candidate solutions in the pool, when picking the next parent solution.

20.3.1.1 *Age-Based Lexicase*

We augment lexicase selection to maintain newly introduced lineages by adding an “age” case to the set of training cases used for selection. We calculate age for an individual as the difference between the current generation and the generation in which its lineage was first injected into the population. In this work, we refer to all non-age-based training cases as “performance-based” training cases. The age case is included (and shuffled) with problem-specific performance-based training cases for each parent selection event. When the age case is encountered, it removes all but the youngest candidate solutions (i.e., those most recently introduced) from the pool of eligible candidate solutions.

Because lexicase selection randomly shuffles the set of training cases for each selection event, the age case may fail to appear early in the shuffle for any given parent selection event, especially when there are a large number of performance-based test cases. This absence could result in the rapid loss of all newly introduced lineages, as these candidate solutions may not yet be competitive on performance-based training cases (as compared to candidates from older lineages). To address this potential issue and emphasize the role of age in this study, we configured the age case to appear within the first 10 training cases (in a random position) for each selection event. This modification ensures that the age-based training case will appear first in lexicase’s shuffle for approximately 10% of selection events each generation, dramatically reducing the chance that all newly injected genetic material will be lost.

20.3.2 *Genetic Programming Experiments*

We assessed age-based lexicase on five program synthesis benchmark problems: Bouncing Balls, Dice Game, GCD, Grade, and Snow Day. For each, we compared

the problem-solving success of random injection with age-based selection, random injection without age-based selection, and a no-injection control (standard lexicase).

20.3.2.1 Genetic Programming System

For all GP experiments, we ran 30 replicates of each treatment. In each replicate, we evolved a population of 1,000 lineage genetic programs for 10,000 generations using the SignalGP representation [27]. Our supplemental material (see [41]) contains the complete instruction sets used for each problem (including source code).

For injection-enabled treatments, we introduced 100 new individuals every 100 generations. On generations where we injected 100 new individuals, we held population size constant by producing 100 fewer offspring via normal selection and reproduction. Any injected candidate solutions were assigned an age of 0, and offspring inherited their parent's age plus one. In exploratory supplemental experiments (included in supplemental material), we evaluated additional injection intervals (20 and 200 generations), finding no single interval was consistently best for all problems and injection modes [41]. We generated random programs to inject by first choosing a program size at random between valid minimum and maximum program lengths (1 and 128 instructions, respectively). For each instruction position in the program, we selected a random instruction (with uniform probability) and random instruction arguments (with uniform probability).

We reproduced programs asexually, applying mutations to offspring. We applied single-instruction substitutions, insertions, and deletions, each at a per-instruction rate of 0.1%. We applied single-argument substitutions at a 0.1% per-argument rate, and we applied slip mutations [28], which can duplicate or delete sequences of instructions, at a per-offspring rate of 5%. For all problems, we limited programs to a maximum length of 128 instructions. When evaluating a program on a training case, we allowed a maximum of 128 time steps (i.e., instruction-executions) to produce output.

20.3.2.2 Program Synthesis Problems

The program synthesis benchmark suites contain problems adapted from common introductory programming problems [11, 13]. These problems are well-studied and frequently used to benchmark GP systems. Brief descriptions of each problem used in this study are given below:

- **Bouncing Balls** [11]: Given three inputs describing a starting height (float in [1.0, 100.0]), a height after the initial bounce of a dropped ball (float in [1.0, 100.0]), and a number of bounces (integer in [1, 20]), the program must compute and return the total distance that the dropped ball would have traveled across all bounces.

- **Dice Game [11]:** Given two integer inputs (in [1 and 1000]), n and m , that represent the number of sides on each of two dice, the program must return the probability that an n -sided die rolls strictly higher than an m -sided die.
- **GCD [11]:** Given two integers in range [1, 1000000], the program must output the largest integer that evenly divides into each.
- **Grade [13]:** Programs receive a *score* and five monotonically decreasing integer inputs (in [0, 100]) that represent unique “grade” thresholds describing the minimum score to receive an *A*, *B*, *C*, or *D* grade. The program must output the appropriate letter grade for the *score* given the threshold values or return *F* if *score* < *D*.
- **Snow Day [11]:** Programs receive four inputs, including a number of hours (integer in [0, 20]), how much snow is on the ground (float in [0.0, 20.0]), the rate of snow fall (float in [0.0, 10.0]), and the proportion of snow melting per hour (float in [0.0, 1.0]). The program must output the amount of snow on the ground after the given number of hours elapses.

For each problem, we evaluated candidate solutions with a set of 100 training cases, and we used a set of 1,000 testing cases to determine problem-solving success. All training and testing sets were generated using data provided by benchmark suite authors [11, 13]. The training and testing sets used in our experiments are available in our supplemental material [41]. We categorized a replicate as successful if it produced a program capable of solving all training and testing cases. For all problems, we included input-output edge cases in both the training and testing sets to ensure adequate assessment of generalization.

20.3.3 Sudoku Puzzle Construction

We further assessed age-based lexicase on a different type of problem: evolving a Sudoku puzzle that is evaluated based on the solving experience that it provides to the player. We then evolved puzzles with the goal of finding the hardest solving experience, as well as producing particular final states in the puzzles.

20.3.3.1 Puzzle Evaluation and Evolution Methodologies

We chose 12 distinct solving techniques that are commonly used by humans (described below) and ranked these from easiest to hardest. When a puzzle is evaluated, we first try the easiest technique and identify all cells where the rule can be applied. If it cannot be applied, we move on to the second easiest technique, then the third, and so on, until we find an actionable technique (or identify that no further progress can be made). Once a technique is chosen, we apply all instances of it and append it onto the “solving profile” for this puzzle. When evaluation is complete for a puzzle, we have a solving profile that lists out the easiest techniques that can be

applied, in order, and rates that puzzle as “solved” if a unique solution was reached, “unsolved” if evaluation stopped early due to no more applicable techniques, or “invalid” if a contradiction was found.

While lexicase selection usually operates on a set of input/output test cases, we instead used tests based on the starting puzzle and its solving profile. Specifically, we used:

- Eleven tests for individual solving techniques, based on how many times each technique appeared in the solving profile (we did not include one for technique zero since it is trivial)
- One test based on whether the puzzle could be solved (where unsolved puzzles were ranked above invalid puzzles).
- One test based on the total “length” of the solving profile (i.e., the number of rounds of techniques needed to solve the puzzle).
- One test based on the overall “difficulty” of a puzzle, calculated as the sum of technique difficulties in the solving profile (if a given technique appears more than once, each instance is included.)
- One test based on the “variety” of techniques needed to solve a puzzle, calculated as the number of distinct techniques that appear in the profile.
- One test based on the “sparseness” of a puzzle, calculated as the number of empty cells in its starting state.

For a grand total of 16 tests used by default. In treatments where we use age-based lexicase, we also added an associated test case, which we guaranteed to appear in the first set of ten (as above).

We started all evolving populations with a set of empty puzzles, with a population size of 1000. On birth, each puzzle had a 0.015 per-cell chance of mutation, for an expected number of mutations of $0.015 * 81 = 1.215$; approximately 29% of offspring had no mutations, a 36% chance of one mutation, and a 35% chance of multiple mutations. A mutation would change a cell to the values 1 through 9 or “empty” with equal probability. Every 500 generations, we injected 100 new, empty puzzles, which were given an advantage only in the age-based treatments. Overall, we allowed each run to evolve for 10,000 generations.

20.3.3.2 Sudoku Solving Techniques

Each cell in a Sudoku puzzle tracks if a value is set for that cell and, if not, what values remain possibilities. Initially, cells are either set by the starting state or all values are included as possibilities. The rules below either set cells or reduce the possibilities available to a given cell. If ever a cell is not set and has no remaining possibilities, the puzzle is identified as “invalid”.

The solving techniques available, in order, are:

0. If a cell has been set to a value, remove that value as a option from all other cells in shared regions.

1. **Last Cell Option:** If a cell has only one option remaining, set it to that option.
2. **Last Region Option:** If a region has only one cell left with a given option, set that cell to the option.
3. **Overlapping regions:** If all options for a state in region A are also in region B, remove that state as an option in other cells of region B.
4. **Conjugate pairs:** In region A, if two cells are limited to the same two states, eliminate those two states from everywhere else in A.
5. **Hidden pairs:** If two cells are the only ones with two states in a region, eliminate all other options from those two cells.
6. **X-Wing:** If state S can only be in the same two positions in for two different rows, eliminate S from elsewhere in the associated columns (also, swap rows/cols)
7. **X-Wing+Boxes:** Similar to X-Wing, but where one of the regions can be a 3×3 box.
8. **Conjugate triples:** Same as a conjugate pairs, but with 3 cells and 3 states.
9. **Hidden triples:** Same as hidden pairs, but with 3 cells and 3 states.
10. **Swordfish:** Same as X-Wing, but with three positions in each of three rows/cols.
11. **Swordfish4:** Same as X-Wing, but with four positions in each of four rows/cols.

Many other solving techniques are also possible in Sudoku puzzles, but for this work, we limited ourselves to the options above.

20.3.4 Statistical Analyses

For all analyses, we compared measurements taken from different selection treatments on the same problem. When comparing problem-solving success rates for program synthesis problems, we used pairwise Fisher's Exact tests (significance level of 0.05). When comparing multiple distributions of measurements taken from different treatments, we first applied a Kruskal-Wallis test to screen for statistical differences among independent conditions. For comparisons in which the Kruskal-Wallis test was significant, we used post-hoc Wilcoxon rank-sum tests to identify pairwise differences. Where appropriate, we corrected for multiple comparisons using a Holm-Bonferroni correction. Complete statistical analysis results are available in our supplemental material [41].

20.4 Results

20.4.1 Program Synthesis Problem-Solving Success

Figure 20.1 shows the problem-solving success across all five program synthesis problems for each of mode of selection that we tested: random injection with age-based selection, random injection without age-based selection, and standard lexicase

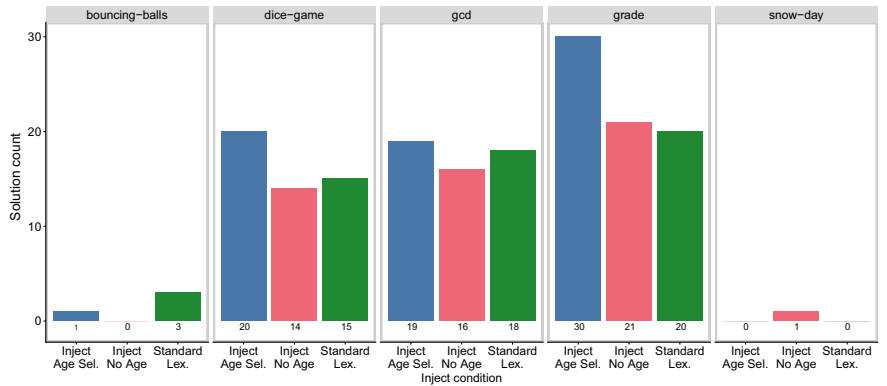


Fig. 20.1 Problem-solving success across five program synthesis problems for three selection modes: random injection with age-based selection (Inject Age Sel.), random injection without age-based selection (Inject No Age), and a standard lexicase control (Standard Lex.). The number of successful replicates for a given treatment is given below the associated bar

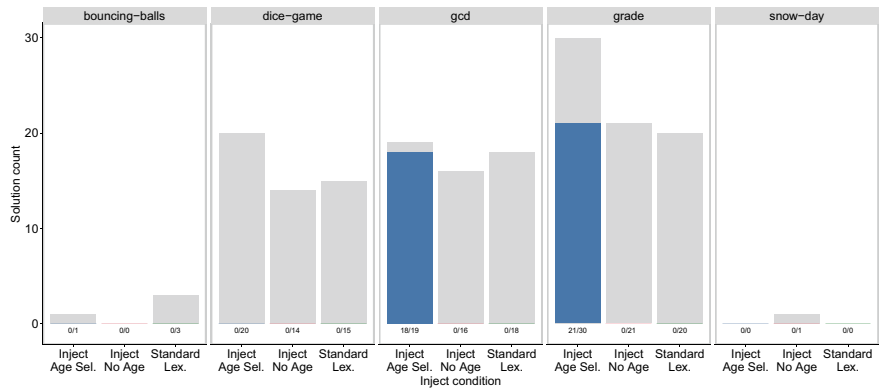


Fig. 20.2 Proportion of solutions that descend from candidate solutions injected into the population for each selection treatment. The light-gray background bars indicate overall problem-solving success for a given treatment (as in Fig. 20.1), and the text below each bar gives the associated proportion. No new programs are injected in the Standard Lex. Treatment, so it is not possible for a solution to descend from an injected program

selection. We consider a replicate successful if it produces a program capable of solving all training and testing cases.

Age-based selection with random injection performed significantly better than each other selection treatment for the Grade problem (corrected pairwise Fisher’s exact tests, $p < 0.005$), and we detected no significant differences in problem-solving success among selection treatments for all four other problems. This result is consistent with supplemental experiments that we ran with different injection intervals [41].

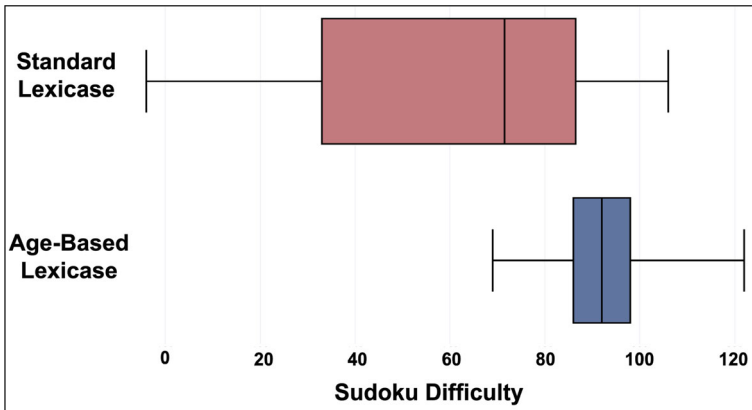


Fig. 20.3 Boxplot distributions illustrating the maximum difficulty for solvable Sudoku puzzles produced from each run in each treatment (40 runs total per treatment). Horizontal lines indicate the full range of the data. Boxes extend from the 25th percentile to the 75th percentile. The vertical line in each box indicate the median of each distribution. Negative difficulty levels indicate the no solvable puzzles were produced

To determine if injected lineages contributed to the problem-solving success when using age-based selection with random injection, we analyzed the proportion of evolved solutions that descended from injected individuals, which is shown in Fig. 20.2. Without selection for age, we do not expect the lineages of randomly injected programs to survive to eventually produce a solution. Indeed, across all problems, no solutions from treatments with injection but no age-based selection descended from injected programs. When using age-based selection, the majority of solutions that evolved for the GCD and Grade problems descended from injected programs (18/19 and 21/30, respectively). For each of the three other problems, however, no evolved solutions descended from injected programs; that is, all solutions to these problems descended from the original populations.

20.4.2 Sudoku Puzzle Generation Success

Our lexicase runs all produced a diverse sampling of Sudoku puzzles. For the purpose of this comparison, we focused on the highest difficulty level of solvable puzzles to come out of each run. Figure 20.3 shows the comparison in difficulty between the final populations with and without lineage age as a lexicase test.

Age-based selection was able to produce significantly more difficult puzzles than standard lexicase (pairwise Fisher’s exact test, $p < 0.005$). While injections occurred every 500 generations in all runs, new lineages were supported by selection only in age-based lexicase.

Fig. 20.4 The Sudoku puzzle with the maximum difficulty (122) from any of the evolution runs

3		2						
6								
9			2	8		3		
	5				8		7	
			6					5
			9	4	5			
	7			1		2		
		1		5			6	
		3	8			9		

The final most-difficult board in the standard lexicase populations never arose from an injected lineage, while they did come from an injected lineage in 36 out of the 40 runs (90%) with age-based lexicase (with the latest being injected at generation 8500 of 10,000). The overall most difficult evolved puzzle arose in one of the age-based lexicase runs and is displayed in Fig. 20.4. This result emphasizes the importance of starting position for this Sudoku-generation problem, which would likely be locked in by the first few mutations to occur.

20.5 Conclusion

Our experiments suggest that adding an age-focused test case to lexicase selection, along with injections of random solutions, can be effective in some problem domains, but not all. Age-based selection with random injection improved problem-solving success for one out of the five program synthesis problems we tested, but did not perform significantly different than standard lexicase selection on each of the other four problems. The majority of solutions for the GCD and Grade problems (18/19 and 21/30) descended from injected programs. For the Sudoku construction problem, selecting for age alongside random injections allows us to generate puzzles with higher difficulty. We speculate that the Grade problem and the Sudoku construction problem benefit from exploring new regions in the search space, which new starting points gives them access to. This characteristic makes them particularly suitable for age-based lexicase.

Overall, our results motivate further investigations to identify scenarios where age-driven diversity will be most helpful and how to configure runs to meet these goals. For example, future work should investigate the frequency of injecting random solutions, the number of solutions to inject, the population size, and the best way of

augmenting lexicase to support the injection of new solution lineages (e.g., an age training case or age population layers as in [10]). Future work will use Hernandez et al.'s suite of diagnostic fitness landscapes to study how these different ways of configuring age-based lexicase interact [16]. Initial investigations under these diagnostics, available in our supplemental material [41], suggest that auxiliary age-driven diversity can be counterproductive in lexicase selection in some scenarios. Population size directly mediates the number of “specialist” lineages (i.e., lineages that specialize on different subsets of training cases) that lexicase can reliably maintain over many generations [7]. Augmenting lexicase selection with an age case adds an additional niche that lineages can specialize on, potentially reducing the number of niches comprising problem-related training cases that lexicase can support. Thus, after an injection, we gain new diversity from new lineages, but these newly injected individuals may act as an invasive species, driving established specialists (descended from older lineages) extinct. Because the descendants of newly injected individuals are unlikely to perform well on a problem for many generations, the overall fitness of the population may degrade with each injection event. In future work, we plan to address this issue by estimating the current number of distinct niches occupied in the population and use this information to dynamically adjust population size or manipulate shuffles for training cases during selection to avoid losing useful diversity after injecting new individuals into a population.

Other potential avenues for this work include incorporating age-based diversity into quality diversity algorithms and exploring other modes of injection. So far, we have injected only random solutions. Current ideas for novel modes of injection include random recombination injection and complementary recombination injection. Random recombination injection generates new solutions by combining two randomly selected individuals, whereas complementary recombination injection chooses the best pairings of current candidate solutions based on a collective performance metric. Preliminary results for both injection techniques are included in our supplemental material [41].

Acknowledgements We thank the participants of Genetic Programming Theory and Practice XXI for helpful comments and suggestions on our work. This work was also supported in part through computational resources and services provided by the Institute for Cyber-Enabled Research at MSU.

References

1. Aenugu, S., Spector, L.: Lexicase selection in learning classifier systems. In: Proceedings of the Genetic and Evolutionary Computation Conference—GECCO '19, pp. 356–364. ACM Press, Prague, Czech Republic (2019). <https://doi.org/10.1145/3321707.3321828>. <http://dl.acm.org/citation.cfm?doid=3321707.3321828>
2. Belding, T.C.: The distributed genetic algorithm revisited. In: Proceedings of the 6th International Conference on Genetic Algorithms, pp. 114–121. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1995)
3. Conceição António, C.A.: A hierarchical genetic algorithm with age structure for multimodal optimal design of hybrid composites. *Struct. Multidiscip. Optim.* **31**(4), 280–294 (2006).

- <https://doi.org/10.1007/s00158-005-0570-9>
4. De Jong, K.A.: An analysis of the behavior of a class of genetic adaptive systems. Ph.D. thesis, USA (1975). AAI7609381
 5. Deb, K., Agrawal, S.: Understanding interactions among genetic algorithm parameters. In: *Foundations of Genetic Algorithms*, vol. 5, pp. 265–286 (1998). <https://api.semanticscholar.org/CorpusID:686145>
 6. Ding, L., Spector, L.: Optimizing neural networks with gradient lexicase selection. In: *International Conference on Learning Representations* (2022). https://openreview.net/forum?id=J_2xNmVcY4
 7. Dolson, E.L., Banzhaf, W., Ofria, C.: Ecological theory provides insights about evolutionary computation. preprint, *PeerJ Preprints* (2018). <https://doi.org/10.7287/peerj.preprints.27315v1>. <https://peerj.com/preprints/27315>
 8. Gabriel, W., Lynch, M., Bürger, R.: Muller’s ratchet and mutational meltdowns. *Evolution* **47**(6), 1744–1757 (1993). <https://doi.org/10.1111/j.1558-5646.1993.tb01266.x>. <https://academic.oup.com/evolut/article/47/6/1744/6870416>
 9. Galan, S.F., Mengshoel, O.J.: Generalized crowding for genetic algorithms. In: *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, GECCO ’10*, pp. 775–782. Association for Computing Machinery, New York, NY, USA (2010). <https://doi.org/10.1145/1830483.1830620>
 10. Helmuth, T., Abdelhady, A.: Benchmarking parent selection for program synthesis by genetic programming. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, GECCO ’20*, pp. 237–238. Association for Computing Machinery, New York, NY, USA (2020). <https://doi.org/10.1145/3377929.3389987>
 11. Helmuth, T., Kelly, P.: PSB2: the second program synthesis benchmark suite. In: *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 785–794. ACM, Lille France (2021). <https://doi.org/10.1145/3449639.3459285>. <https://dl.acm.org/doi/10.1145/3449639.3459285>
 12. Helmuth, T., McPhee, N.F., Spector, L.: Effects of lexicase and tournament selection on diversity recovery and maintenance. In: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion—GECCO ’16 Companion*, pp. 983–990. ACM Press, Denver, Colorado, USA (2016). <https://doi.org/10.1145/2908961.2931657>. <http://dl.acm.org/citation.cfm?doid=2908961.2931657>
 13. Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference—GECCO ’15*, pp. 1039–1046. ACM Press, Madrid, Spain (2015). <https://doi.org/10.1145/2739480.2754769>. <http://dl.acm.org/citation.cfm?doid=2739480.2754769>
 14. Helmuth, T., Spector, L., Matheson, J.: Solving uncompromising problems with lexicase selection. *IEEE Trans. Evol. Comput.* **19**(5), 630–643 (2015). <https://doi.org/10.1109/TEVC.2014.2362729>. <http://ieeexplore.ieee.org/document/6920034/>
 15. Hernandez, J.G., Lalejini, A., Ofria, C.: An exploration of exploration: measuring the ability of lexicase selection to find obscure pathways to optimality. In: Banzhaf, W., Trujillo, L., Winkler, S., Worzel, B. (eds.) *Genetic Programming Theory and Practice XVIII*, pp. 83–107. Springer Nature, Singapore (2022). https://doi.org/10.1007/978-981-16-8113-4_5
 16. Hernandez, J.G., Lalejini, A., Ofria, C.: A suite of diagnostic metrics for characterizing selection schemes (2023). <https://doi.org/10.48550/arXiv.2204.13839>
 17. Hernandez, J.G., Saini, A.K., Moore, J.H.: Lexicase Selection Parameter Analysis: Varying Population Size and Test Case Redundancy with Diagnostic Metrics (2024). *ArXiv:2407.15056 [cs]*
 18. Hornby, G.S.: ALPS: The age-layered population structure for reducing the problem of premature convergence. In: *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, GECCO ’06*, pp. 815–822. Association for Computing Machinery, New York, NY, USA (2006). <https://doi.org/10.1145/1143997.1144142>

19. Hornby, G.S.: Steady-state alps for real-valued problems. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, GECCO '09, pp. 795–802. Association for Computing Machinery, New York, NY, USA (2009). <https://doi.org/10.1145/1569901.1570011>
20. Hornby, G.S.: A Steady-state version of the age-layered population structure EA. In: Riolo, R., O'Reilly, U.M., McConaghy, T. (eds.) Genetic Programming Theory and Practice VII, pp. 87–102. Springer US, Boston, MA (2010). https://doi.org/10.1007/978-1-4419-1626-6_6
21. Hu, J., Goodman, E., Seo, K., Fan, Z., Rosenberg, R.: The hierarchical fair competition (HFC) framework for sustainable evolutionary algorithms. *Evol. Comput.* **13**(2), 241–277 (2005). <https://doi.org/10.1162/1063656054088530>
22. Huber, A., Mlynski, D.: An age-controlled evolutionary algorithm for optimization problems in physical layout. In: ISCAS '98. Proceedings of the 1998 IEEE International Symposium on Circuits and Systems (Cat. No.98CH36187), vol. 6, pp. 262–265 (1998). <https://doi.org/10.1109/ISCAS.1998.705261>
23. Hutter, M.: Fitness uniform selection to preserve genetic diversity. In: Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600), vol. 1, pp. 783–788 (2002). <https://doi.org/10.1109/CEC.2002.1007025>
24. Kubota, N., Fukuda, T.: Genetic algorithms with age structure. *Soft Comput.-Fusion Found. Methodol. Appl.* **1**(4), 155–161 (1997). <https://doi.org/10.1007/s005000050017>
25. La Cava, W., Spector, L., Danai, K.: Epsilon-lexicase selection for regression. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016, pp. 741–748. ACM, Denver Colorado USA (2016). <https://doi.org/10.1145/2908812.2908898>. <https://dl.acm.org/doi/10.1145/2908812.2908898>
26. Lalejini, A., Dolson, E., Vostinar, A.E., Zaman, L.: Artificial selection methods from evolutionary computing show promise for directed evolution of microbes. *eLife* **11**, e79,665 (2022). <https://doi.org/10.7554/eLife.79665>. <https://elifesciences.org/articles/79665>
27. Lalejini, A., Ofria, C.: Evolving event-driven programs with SignalGP. In: Proceedings of the Genetic and Evolutionary Computation Conference on—GECCO '18, pp. 1135–1142. ACM Press, Kyoto, Japan (2018). <https://doi.org/10.1145/3205455.3205523>. <http://dl.acm.org/citation.cfm?doid=3205455.3205523>
28. Lalejini, A., Wiser, M.J., Ofria, C.: Gene duplications drive the evolution of complex traits and regulation. In: Proceedings of the 14th European Conference on Artificial Life ECAL 2017, pp. 257–264. MIT Press, Lyon, France (2017). https://doi.org/10.7551/ecal_a_045. https://www.mitpressjournals.org/doi/abs/10.1162/isal_a_045
29. Lehman, J., Stanley, K.O.: Abandoning objectives: evolution through the search for novelty alone. *Evol. Comput.* **19**(2), 189–223 (2011). https://doi.org/10.1162/EVCO_a_00025
30. Matsumoto, N., Saini, A.K., Ribeiro, P., Choi, H., Orlenko, A., Lyytikäinen, L.P., Laurikka, J.O., Lehtimäki, T., Batista, S., Moore, J.H.: Faster convergence with lexicase selection in tree-based automated machine learning. In: Pappa, G., Giacobini, M., Vasecek, Z. (eds.) Genetic Programming, vol. 13986, pp. 165–181. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-29573-7_11. https://link.springer.com/10.1007/978-3-031-29573-7_11. Series Title: Lecture Notes in Computer Science
31. Metevier, B., Saini, A.K., Spector, L.: Lexicase selection beyond genetic programming. In: Banzhaf, W., Spector, L., Sheneman, L. (eds.) Genetic Programming Theory and Practice XVI, pp. 123–136. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-04735-1_7. http://link.springer.com/10.1007/978-3-030-04735-1_7. Series Title: Genetic and Evolutionary Computation
32. Moore, J.M., Stanton, A.: Lexicase selection outperforms previous strategies for incremental evolution of virtual creature controllers. In: Proceedings of the 14th European Conference on Artificial Life ECAL 2017, pp. 290–297. MIT Press, Lyon, France (2017). https://doi.org/10.7551/ecal_a_050. https://www.mitpressjournals.org/doi/abs/10.1162/isal_a_050
33. Mouret, J.B., Clune, J.: Illuminating search spaces by mapping elites (2015)
34. Pozzuoli, A., Ross, B.: Increasing features in map-elites using an age-layered population structure. In: 2023 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8 (2023). <https://doi.org/10.1109/CEC53210.2023.10254093>

35. Pugh, J.K., Soros, L.B., Stanley, K.O.: Quality diversity: A new frontier for evolutionary computation. *Front. Robot. AI* **3** (2016). <https://doi.org/10.3389/frobt.2016.00040>. <https://www.frontiersin.org/articles/10.3389/frobt.2016.00040>
36. Real, E., Aggarwal, A., Huang, Y., Le, Q.V.: Regularized evolution for image classifier architecture search. In: *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33(01), pp. 4780–4789 (2019). <https://doi.org/10.1609/aaai.v33i01.33014780>
37. Sareni, B., Krahenbuhl, L.: Fitness sharing and niching methods revisited. *IEEE Trans. Evol. Comput.* **2**(3), 97–106 (1998). <https://doi.org/10.1109/4235.735432>
38. Schmidt, M., Lipson, H.: Age-fitness pareto optimization. In: Riolo, R., McConaghy, T., Vladislavleva, E. (eds.) *Genetic Programming Theory and Practice VIII*, pp. 129–146. Springer, New York, NY (2011). https://doi.org/10.1007/978-1-4419-7747-2_8
39. Spector, L.: Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In: *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, pp. 401–408. ACM, Philadelphia Pennsylvania USA (2012). <https://doi.org/10.1145/2330784.2330846>
40. Spector, L., Moscovici, E.: Recent developments in auto constructive evolution. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 1154–1156. ACM, Berlin Germany (2017). <https://doi.org/10.1145/3067695.3082058>. <https://dl.acm.org/doi/10.1145/3067695.3082058>
41. Suzue, K., Ofria, C., Lalejini, A.: Supplemental material (Archived GitHub repository) (2024). <https://doi.org/10.5281/zenodo.12751839>. GitHub at <https://github.com/amlalejini/age-based-lex>
42. Willis, A., Patel, S., Clack, C.D.: GP age-layer and crossover effects in bid-offer spread prediction. In: *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, GECCO '08*, pp. 1665–1672. Association for Computing Machinery, New York, NY, USA (2008). <https://doi.org/10.1145/1389095.1389407>
43. Wu, Z., Yang, Q., Wang, X., Xu, D., Zhao, J., Li, J.: Preset model of bending force in 6-high universal crown tandem cold rolling mill based on symbolic regression. *Ironmak. Steelmak.* **50**(11), 1668–1682 (2023). <https://doi.org/10.1080/03019233.2023.2218777>

Index

A

Active learning, 297, 298, 304, 305, 316
Aesthetic assessment, 322
Algorithm configuration, 265, 375
Algorithmic art, 322
Architecture, 68, 70, 75, 78, 81
Artificial intelligence, 358
AutoEncoders (AEs), 107, 108, 110, 112–115, 118, 121, 124–127, 129–131
Autonomous control, 338
Autonomous control systems, 339
Autonomous systems, 354

B

Baldwin effect, 259, 266, 271
Ball mutation, 238, 240–243, 256
Benchmark suite, 375
Bidirectional Encoder Representations from Transformers (BERT), 51, 57–60, 62–64
Biomedical data, 297, 299, 304
Bloat, 237
Boolean gates, 219–223, 225
Boolean networks, 219–222, 225–227, 229, 231, 234

C

Cartesian genetic programming, 200–203, 205, 207–209, 212, 213, 215, 299
Circuit evolution, 220
Classification, 27–30, 36–38, 40, 41, 44, 46
Classifier, 321, 323, 327–329, 331–333
Code building genetic programming, 133
Concomitant evolution, 177, 182
Content addressable memory, 200

Context-free grammar genetic programming, 180
Context-free grammars, 178–183, 189–191, 193, 195
Control systems, 338
Convergence analysis, 260
Convolutional neural network, 199–206, 210, 212–217, 322, 323, 326, 327, 331, 332
Crossover, 51–57, 62, 64

D

Data handling, 3, 16
Data modeling, 1, 3, 12, 23
Data sampling, 299
Deep learning, 28, 29, 43, 51, 52, 64, 152, 153, 298, 303
Deep Neural Networks (DNNs), 107, 108, 110
Deep reinforcement learning, 51, 52
Deflate mutation, 238, 241–245, 256
Diagnostics, 377, 378, 381–383, 391
Digital design, 275, 288, 290
Diversity maintenance, 395–397
Dynamically structured grammatical evolution, 177, 178, 180

E

Elfa, 337
Energy efficiency, 202, 276
Equivalence class, 149
Evaluation of explainability, 365
Evaluation of interpretability, 364
Evolutionary algorithms, 299
Evolutionary Machine Learning (EML), 108

Explainability, 365
 Explainable artificial intelligence, 357, 358
 Exploitation, 375, 376, 378, 379, 381–385, 390, 391
 Exploration, 375, 377, 378

F

Feature construction, 27, 29–31, 33, 34, 36, 40, 42–44, 46
 Feature engineering, 27–29, 31, 34, 35, 46
 Feature generation, 36
 Fidelity, 357, 362–365, 369
 Field-Programmable Gate Array (FPGA), 275–278, 280–291
 Fitness comparison, 322
 Fitness landscapes, 153, 156, 322
 Floating-point-based GP, 275, 277

G

Generative art, 321–324, 328, 329, 333
 Gene regulatory networks, 340
 Genetic improvement, 321, 322
 Genetic programming, 299
 Genetic programming with memory, 199–201
 Genotype-phenotype mapping, 151, 162, 166
 Geometric Semantic Genetic Programming (GSGP), 237–240, 245, 249, 251, 254–256
 GP supporting LLM, 69, 78, 82
 Gradient descent, 259, 262–268, 271
 Grammar-guided genetic programming, 177–180, 182, 195
 Grammatical evolution, 177–180, 182, 322, 325
 Graph-based GP, 298

H

Hardware acceleration, 276
 Hardware accelerator, 199, 200, 202, 203, 216
 Hardware/software co-design, 275
 High-performance computing, 277, 328

I

Image analysis, 322
 Image optimization, 322
 Image processing, 297–302, 305

Image reconstruction, 107–109, 112, 114, 117, 118, 120, 121, 124, 127, 131
 Inflate mutation, 238, 241–243, 245, 256
 Interpretability, 299, 365
 Interpretable machine learning, 298, 299, 314, 358

L

Lamarckian learning, 260
 Large language model, 67–75, 77–83
 as part of GP, 69
 supported by GP, 69
 supporting GP, 69
 Lexicase selection, 20, 326, 375–382, 384–387, 389–391, 395–397, 399, 400, 403, 405, 407, 408
 Lineage age, 395, 406
 LLM and GP, 79, 81
 LLM as part of GP, 73–75, 82
 LLM supporting GP, 79, 82
 Local search, 259–261, 264–269, 271

M

Modeling assumptions, 2, 23
 Multimodality, 299
 Multiplier problem, 230
 Multiply and Accumulate (MAC), 203
 Multitask learning, 87–89, 91, 97
 Multi-tree genetic programming, 29
 Mutation, 51, 52, 57–64

N

NeuroEvolution (NE), 107
 Non-encoding nodes, 177, 182, 185–189, 195

O

Operon, 277, 281, 283, 285, 286
 Overfitting avoidance, 254

P

Parent selection, 375–378, 382, 390
 Pareto tournament, 20–22, 24
 Pareto tournament selection, 20
 Parity problem, 230, 231
 Partially observable systems, 88, 100
 Performance measures, 12
 Predictive control, 88, 93
 Premature convergence, 395–398

Problem decomposition, 88, 92, 94, 95
Program semantics, 141
Program synthesis, 133, 375, 377, 395, 397,
399–401, 404, 407
Prompt, 67–74, 77–82
PushGP, 133

R

Random rewiring, 219
Ratcheted random search, 219, 227–230
Reinforcement learning, 52, 53, 55, 59, 67,
75, 88, 89, 91
Representation, 1, 3, 7, 11, 16, 18, 23, 24
Runtime cost, 148

S

Search space exploration, 396
Selection, 1, 7, 14, 17, 19–22, 24
Selection scheme, 375, 376, 378–380, 383,
390
Semantic learning, 237
Semantic Learning algorithm based on
Inflate and deflate Mutations

(SLIM), 237–239, 241–245, 249,
255, 256
Semantics, 151–154, 158, 159, 167, 171
SHAP, 361–365, 369, 370
Sharpness, 151–153, 155–160, 162, 167,
171–173
Sudoku, 395, 397, 399, 402–404, 406, 407
Symbolic regression, 1, 3, 4, 6, 8–12, 14,
18, 20, 22, 23, 61, 75, 152, 179, 200,
201, 205, 206, 239, 259–262, 305,
357, 360, 378, 399

T

Tabular data, 27–30, 42, 43
Tangled program graphs, 87, 88, 90–92,
94–96, 98–101
Temporal memory, 88, 92
Time series forecasting, 87–89, 94, 95

W

Weight compression, 203