Reconsideration and Extension of Cartesian Genetic Programming

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik
von

Roman Tobias Kalkreuth

Dortmund, den 08.03.2021

Tag der mündlichen Prüfung:
27.07.2021

Dekan:
Prof. Dr.-Ing. Gernot A. Fink
Prodekan:
Prof. Dr. Peter Bucholz

Gutachter:
Prof. Dr. rer. nat. Günter Rudolph, Technische Universität Dortmund
Jun.-Prof. Dr. rer. nat. Paul Kaufmann, Johannes Gutenberg-Universität Mainz

Acknowledgements

Well, as time passes by, the last years have been an amazing and extraordinary journey which finally resulted into something of which I feel proud of. Within the framework of my doctoral thesis I feel grateful that I had the chance to make a contribution to the field of graph-based Genetic Programming. Obviously, it wouldn't be a doctoral thesis if there would be no challenges and struggles involved. Therefore, I also feel very grateful for the people who supported me over the last years:

Günter, I never told you that people at my former university of applied sciences doubted me when I told them about my doctoral project in Dortmund. Especially after they saw your publication list. ;-) However, I guess I proved them wrong. Anyway, you have been very supportive to people from universities of applied sciences over the last years and that is somethng I would like to praise here!

Paul, I am very happy and grateful that we met in Vancouver at the WCCI 2016! Your feedback was always helpful and it was overall very supportive to have a peer within the graph-based Genetic Programming community! I am personally looking forward to continuing crushing several graph-based GP issues with you!

A **big** thanks goes to my family, especially to my mother and my brother!

Helena, I guess you feel very happy to read this page because it is a significant indicator that my doctoral project is finished. You supported me so much in the final and most demanding phase of the whole project. I guess there is clear evidence that the Pareto principle also holds for dissertations. ;-)

Georg, your support was such a big help over the past few years. Offering me such a positive and unique possibility to retreat from stressfull phases and balancing my scientific work with learning to play piano.

Andre and Nikki, you both have been such supportive and amazing working colleagues! Moreover, we spent very good times together outside of the academic realm and I am grateful to have met you both!

Gundel, what shall I say? I think you occupied a special position at our chair for lots of people. I am pretty sure that without our humorous conversations, my doctoral project would have been definetly more challenging.

I also want to thank **Prof. Dr. Jörg Krone**. As the supervisor of my master thesis you introduced me to the field of Genetic Programming. You have been also very supportive over the last years to my doctoral project by giving advices.

My thanks goes also to **Prof. Dr. Heinrich Müller** and **Prof. Dr. Peter Bu-**

cholz for their support of my project as members of the committee.

Dominik Kopczynski, Simon Wessing, Jan Quadflieg, Boris Naujoks, Denis Kurz, Vanessa Volz, Christine Dahn, Fritz Bökler, Florian Kurpicz, Christine Dahn, Christiane Spisla, Marianna D'Addario, Nils Kriege, Till Schäfer, Cindy Konen, Marie-Theres Sperner, Sebastian Schäfer, Jurij Kuzmic, Marco Pleines, Marius Bommert, Igor Vatolkin, Mike Preuss, Bernd Zey, Arno Pasternak, Oliver Flasch, Horst Schirmeier and Alexander Lochmann

You made my time at the University and in Dortmund a very valuable time. I really enjoyed our contact, lunch breaks and last but not least our very serious chats in the *Lehrstuhl* kitchen! ;-)

I wish everyone the best for the future!

”It is known that every science must have its philosophy, and that it cannot make real progress in any other way.”

”If the philosophy of science is neglected her progress will be unreal, and the entire work will remain imperfect.”

Jean-Baptiste Lamarck: *Zoological Philosophy*
Chapter II: *Importance of the Consideration of Affinities*

Contents

1 Preface

1.1 Format

Almost all results of this thesis which I have authored and co-authored have already been published. Some parts of these peer-reviewed publications are used and reproduced literally in this thesis.

1.2 List of Underlying Publications

The following list describes the underlying peer-reviewed publications of this thesis and gives information about the contributions with joint work. The publications are sorted by the year of publication.

Conference papers

[1] Roman Kalkreuth, Günter Rudolph, and Jörg Krone. Improving convergence in cartesian genetic programming using adaptive crossover, mutation and selection. In *IEEE Symposium Series on Computational Intelligence, SSCI 2015, Cape Town, South Africa, December 7-10, 2015*, pages 1415–1422. IEEE, 2015. URL: https://doi.org/10.1109/SSCI.2015.201, doi:10.1109/SSCI.2015.201

The self-adaptive strategy which has been proposed and evaluated in this paper is described in Chapter 6. I contributed 90% of the work to this publication.

[2] Paul Kaufmann and Roman Kalkreuth. An empirical study on the parametrization of cartesian genetic programming. In Peter A. N. Bosman, editor, *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, pages 231–232. ACM, 2017. URL: https://doi.org/10.1145/3067695.3075980, doi:10.1145/3067695.3075980

The cooperation between Paul Kaufmann and myself led to a comprehensive investigation of the so called *1+4 dogma* in CGP. I contributed 50% of the work to this publication. The results of an empirical study on the parametrization of CGP are described in Chapter 5.

[3] Paul Kaufmann and Roman Kalkreuth. Parametrizing cartesian genetic programming: An empirical study. In Gabriele Kern-Isberner, Johannes Fürnkranz, and Matthias Thimm, editors, *KI 2017: Advances in Artificial Intelligence - 40th Annual German Conference on AI, Dortmund, Germany, September 25-29, 2017, Proceedings*, volume 10505 of *Lecture Notes in Computer Science*, pages 316–322. Springer, 2017. URL: https://doi.org/10.1007/978-3-319-67190-1_26, doi:10.1007/978-3-319-67190-1_26

This is an enlarged version of the previous paper and its content was also used in Chapter 5.

[4] Roman Kalkreuth, Günter Rudolph, and Andre Droschinsky. A new subgraph crossover for cartesian genetic programming. In James McDermott, Mauro Castelli, Lukás Sekanina, Evert Haasdijk, and Pablo García-Sánchez, editors, *Genetic Programming - 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings*, volume 10196 of *Lecture Notes in Computer Science*, pages 294–310, 2017. URL: https://doi.org/10.1007/978-3-319-55696-3_19, doi:10.1007/978-3-319-55696-3_19

A new method of crossover for Cartesian Genetic Programming which has been introduced in this publication is described and evaluated in Chapter 7. I contributed 90% of the work to this publication.

[5] Jakub Husa and Roman Kalkreuth. A comparative study on crossover in cartesian genetic programming. In Mauro Castelli, Lukás Sekanina, Mengjie Zhang, Stefano Cagnoni, and Pablo García-Sánchez, editors, *Genetic Programming - 21st European Conference, EuroGP 2018, Parma, Italy, April 4-6, 2018, Proceedings*, volume 10781 of *Lecture Notes in Computer Science*, pages 203–219. Springer, 2018. URL: https://doi.org/10.1007/978-3-319-77553-1_13, doi:10.1007/978-3-319-77553-1_13

The cooperation between Jakub Husa and myself led to a first comparative study on crossover in CGP. I contributed 50% of the work to this publication. The study is described in Chapter 7.

[6] Roman Kalkreuth and Andre Droschinsky. On the time complexity of simple cartesian genetic programming. In Juan Julián Merelo Guervós, Jonathan Garibaldi, Alejandro Linares-Barranco, Kurosh Madani, and Kevin Warwick, editors, *Proceedings of the 11th International Joint Conference on Computational Intelligence, IJCCI 2019, Vienna, Austria, September 17-19, 2019*, pages 172–179. ScitePress, 2019. URL: https://doi.org/10.5220/0008070201720179, doi:10.5220/0008070201720179

This publication presented first theoretical results on the time complexity of

Cartesian Genetic Programming. Two simple test problems have been studied with a $(1 + 1)$-CGP. The content of this publication has been used for Chapter 4. I contributed 80% of the work to this publication.

[7] Roman Kalkreuth. Two new mutation techniques for cartesian genetic programming. In Juan Julián Merelo Guervós, Jonathan M. Garibaldi, Alejandro Linares-Barranco, Kurosh Madani, and Kevin Warwick, editors, *Proceedings of the 11th International Joint Conference on Computational Intelligence, IJCCI 2019, Vienna, Austria, September 17-19, 2019*, pages 82–92. ScitePress, 2019. URL: https://doi.org/10.5220/0008070100820092, doi:10.5220/0008070100820092

Two advanced phenotypic mutation techniques have been proposed and evaluated in this publication. The effects caused by the proposed methods have been also analyzed. The content of this publication has been used for Chapter 8.

[8] Roman Kalkreuth. A comprehensive study on subgraph crossover in cartesian genetic programming. In Juan Julián Merelo Guervós, Jonathan M. Garibaldi, Christian Wagner, Thomas Bäck, Kurosh Madani, and Kevin Warwick, editors, *Proceedings of the 12th International Joint Conference on Computational Intelligence, IJCCI 2020, Budapest, Hungary, November 2-4, 2020*, pages 59–70. SCITEPRESS, 2020. URL: https://doi.org/10.5220/0010110700590070, doi:10.5220/0010110700590070

This publication covers a comprehensive study about the subgraph crossover in comparision to the traditional approach of using Cartesian Genetic Programming with a $(1+\lambda)$ selection strategy. The content of this publication has been used as the basic framework for the study presented in Chapter 9.

Book chapters

[1] Roman Kalkreuth. *An Empirical Study on Insertion and Deletion Mutation in Cartesian Genetic Programming*, pages 85–114. Springer International Publishing, Cham, 2021. URL: https://doi.org/10.1007/978-3-030-70594-7_4, doi:10.1007/978-3-030-70594-7_4

The experiments of the paper *Two New Mutation Techniques for Cartesian Genetic Programming* covered only one problem domain. This work extends the analysis of both mutation techniques with comprehensive experiments in the symbolic regression domain. This work is an extended and revised version of the corresponding conference paper. This work has been accepted for publication as a chapter in the book *Computational Intelligence* which is part of the Springer *Studies in Computational Intelligence* series. The book is currently in the Springer publication process.

2 Introduction

Over twenty years ago, Miller, Thompson, Kalganova, and Fogarty [95, 50] published first concepts on a graph-based representation for Genetic Programming (GP) called Cartesian Genetic Programming (CGP). The first approach to this graph-based GP representation was an encoding model based on a two-dimensional array of functional nodes connected by feed-forward wires of an FPGA device [95, 50]. GP is traditionally used with trees as program representation. However, since the tree representation does not fit for all problems, development in graph-based representations took place since the mid up to late 90s. Among all graph-based GP representations, CGP can be considered as one of the most popular variants when compared to existing graph-based GP systems.

Reviewing the fundamental and most significant work in the field of CGP, some points become clear:

- The state of knowledge in CGP is one-sided. Conclusions, statements and recommendations on fundamental paradigms in CGP are mostly based on experiments with Boolean function problems and are only very little validated in other problem domains. The fundamental work of the last two decades mainly focused on the investigation of essential dogmas of the CGP functionality such as *Redundancy*, *Computational Efficiency*, and *Neutrality*. Experiments for the study of these dogmas led to general claims about useful parametrization patterns, which have not been comprehensively investigated and validated in the past.

- Fundamental concepts of CGP have been merely validated and investigated on an experimental level. The theoretical state of knowledge in CGP has been significantly neglected. GP, in general, suffers from a significant lack of theoretical knowledge when compared to the number of experimental results. A rigorous study of the CGP literature showed that no runtime analysis of CGP algorithms had been done until the work which will be presented in this thesis.

- CGP has a significant lack of knowledge in the field of genetic operators. The development and investigation of recombination operators and advanced mutation techniques has been mostly disregarded. Former introduced recombination operators have only been tested on a small range of simple benchmark problems. Advanced mutation techniques and the use of multiple mutations that have been successful in tree-based GP have not been tested and investigated until the work presented in this thesis.

- CGP has been mainly used and investigated with only one selection scheme over the last two decades. CGP is mostly used mutation-only, and the $1 + \lambda$ strategy has been established as a predominant way to use CGP. However, other selection schemes, which have been found beneficial in the field of evolutionary computation, were only little evaluated and investigated in CGP.
- CGP has been found inferior to another *state-of-the-art*, method for evolving graphs on a set of well-known benchmark problems. This method is called Evolving Graphs by Graph Programming (EGGP) and has been introduced by Atkinson et al. [3]. The reasons for the inferiority are still unknown and have not been investigated in the past. Consequently, there is only little knowledge about the weaknesses and limitation of CGP, which influences the search performance negatively.

The mentioned points make clear that there is still a significant lack of fundamental knowledge in CGP. Furthermore, the development and investigation of advanced techniques for CGP have been significantly neglected. These two points are the primary motivation for this thesis.

2.1 The Current State of Fundamental Scientific Knowledge in Standard CGP

This section is devoted to a brief analysis of the most significant contributions to fundamental knowledge of standard CGP. For this analysis, we divide the field of standard CGP into the following parts: *Theoretical understanding*, *Redundancy*, *Computational efficiency*, *Genetic operators*, *Selection schemes* and *Parametrization*. A more profound survey and analysis of the previous work of each part will be done in the respective chapters of this thesis.

The most fundamental knowledge of standard CGP exists in the fields of *Redundancy* and *Computational efficiency*. In one of the first empirical studies of CGP, Miller [97] analyzed the computational efficiency on Boolean function problems. Miller analyzed and studied the influence of population size on the efficiency of CGP. The most important key finding was that extremely low populations are most effective for the tested problems. The experiments of this study also revealed that the use of recombination reduces computational effort only marginally. This study can be seen as one of the most fundamental works regarding the algorithmic use of CGP. The results of this study motivated Miller et al. [99] to study the behavior of redundancy in standard CGP with very small population size and with the point mutation technique as the sole genetic operator. The outcome of the study on the redundancy of CGP showed that the evolutionary search in CGP benefits from extremely large genotypes and low rates of mutation. The results of Miller's experiments in [97] and [99] can be seen as the origin of the popular use of CGP with a (1+4)-strategy. However, both fundamental studies focused only on Boolean function problems, which are characterized by discrete fitness. Consequently, the understanding of *Computational efficiency* is one-sided, and concepts found in the Boolean domain must be validated in other problem domains. Another crucial point of this subfield is the accuracy of the experiments. Most experiments of the key publication in this subfield were evaluated by Koza's *computational effort measure* (CE), as described in [67]. This performance measure has received significant criticism [113, 12, 85, 109, 8] for its poor accuracy and statistical invalidity. Because of these issues of the CE, McDermott et al. [90] recommended avoiding the use of the CE to achieve better and more accurate benchmark experiments in the field of GP.

Regarding the field of *Genetic operators*, the most fundamental research focused on the standard point mutation operator in combination with the $(1+4)$-CGP by Miller [97]. The findings and results in the subfield of *Computational efficiency* of CGP are based on the experiments with the point mutation operator. However, additional mutation operators have not been developed and investigated. Furthermore, the understanding of recombination in the field of CGP has been mostly neglected in the past. First reports on initial experiments with crossover in CGP have been done by Clegg et al. [13]. The authors reported that standard genotypic crossover techniques failed to improve the performance of CGP. Because of these experiments, they introduced an arithmetic crossover technique that showed good results on one symbolic regression problem. However, after the initial reporting of the arithmetic crossover technique for CGP, no further experiments or investigations on its behavior have been presented in the framework of peer-reviewed publications.

CGP is mostly used with only one selection scheme, which is used with the $(1+\lambda)$ algorithm. This selection scheme seeks for offspring that have the same fitness value as the normally selected individual to preserve diversity in the population. This idea of adapting a genetic drift that yields diverse individuals having equal fitness has been found as highly beneficial for the performance of the $(1+\lambda)$-CGP algorithm by Yu and Miller [166].

Goldman and Punch [32] investigated how CGP's method for encoding directed acyclic graphs and its mutation operator bias the effective length
of individuals and the distribution of inactive nodes
in the genome. Unlike previous work, the experiments showed

> " that CGP has an innate parsimony pressure that makes it very difficult to evolve individuals with a high percentage of active nodes."

Goldman and Punch [32, p. 933]

Moreover, the authors found that this bias is particularly prevalent as the length of an individual increase. These problems are also compounded by CGP's positional biases, which can make some problems effectively unsolvable.

Goldman and Punch [34] also performed an analysis of CGP evolutionary mechanisms and their results on Boolean problems showed that CGP evolves genomes

> "that are highly inactive, very redundant, and full of seemingly useless constants."

Goldman and Punch [34, p. 359]

On some tested problems, the authors observed that less than 1% of the genome was required to encode the evolved solution. Moreover, the authors found that traditional CGP ordering results

> "in large portions of the genome that are never used by any ancestor of the evolved solution."

Goldman and Punch [34, p. 359]

Regarding the *Theoretical understanding* of CGP, the only work has been done by Woodward [165] by investigating functional complexity in CGP on a theoretical level. The work of Woodward is the only theoretical result, which contributed to the understanding of CGP. Moreover, there exists no work which has been done to achieve theoretical knowledge about the time complexity of CGP.

Recently, a comparison between standard CGP and Evolving Graphs by Graph Programming (EGGP) showed that standard CGP is significantly inferior to EGGP on a set of well-known Boolean benchmark problems [3]. However, the exact reasons for the inferiority of CGP are unknown. Furthermore, the results of the comparison between CGP and EGGP call the role of CGP into question. Important analyses in the field of CGP, which address the demonstrated inferiority are still missing.

We surveyed the most fundamental key publications for the mentioned subfield of CGP and classified the state of knowledge empirically in each subfield. The list of key publications is given in Table 2.1. Figure 2.1 shows a visualization of the current state of knowledge of CGP. The visualization is based on a survey of the refereed publications. We assessed the state of scientific knowledge by the amount and diversity of the experiments. Other criteria were the reproducibility of the results and future work. The red color means that the state of scientific knowledge is poor. Orange represents that initial work has been done, but that necessary future work has not been followed up afterward. Another criterium for this classification is an one-sided state of scientific knowledge. The green color represents a state of scientific knowledge, which is sufficiently fundamental in the respective field. This means that concepts and dogmas have not been validated in different problem domains. Please note that this evaluation of the state of fundamental knowledge in CGP is empirical and based on the view of the author.

Table 2.1: Key publications of significant subfields representing the state of knowledge in standard CGP.

Subfield	**Title**	**Year**	**Reference**
Redundancy	Cartesian Genetic Programming	2000	Miller and Thompson [100]
	The Advantages of Landscape Neutrality in Digital Circuit Evolution	2000	Vassilev and Miller [152]
	Neutrality and the Evolvability of Boolean Function Landscape	2001	Yu and Miller [166]
	What Bloat? Cartesian Genetic Programming on Boolean Problems	2001	Miller [96]
	Finding Needles in Haystacks is not Hard with Neutrality	2002	Yu and Miller [167]
	Finding Needles in Haystacks is Harder with Neutrality	2005	Collins [14]
	Redundancy and Computational Efficiency in Cartesian Genetic Programming	2006	Miller and Smith [99]
	Cartesian Genetic Programming	2011	Miller et al. [98]
	Cartesian Genetic Programming: Why No Bloat?	2014	Turner and Miller [145]
	Analysis of Cartesian Genetic Programming's Evolutionary Mechanisms	2015	Goldman and Punch [34]
	Neutral genetic drift: an investigation using Cartesian Genetic Programming	2015	Turner and Miller [150]
Computational Efficiency	An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach	1999	Miller [97]
	Finding Needles in Haystacks is Harder with Neutrality	2005	Collins [14]
	Redundancy and Computational Efficiency in Cartesian Genetic Programming	2006	Miller and Smith [99]
	A New Crossover Technique for Cartesian Genetic Programming	2007	Clegg et al. [13]
	Cartesian Genetic Programming	2011	Miller et al. [98]
	Cartesian Genetic Programming: Why No Bloat?	2015	Turner and Miller [150]
Genetic Operators	An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach	1999	Miller [97]
	Cartesian Genetic Programming	2000	Miller and Thompson [100]
	Positional independence and recombination in Cartesian Genetic Programming	2006	Cai et al. [9]
	A New Crossover Technique for Cartesian Genetic Programming	2007	Clegg et al. [13]
	Length Bias and Search Limitations in Cartesian Genetic Programming	2013	Goldman and Punch. [32]
	ONMCGP: Orthogonal Neighbourhood Mutation Cartesian Genetic Programming for Evolvable Hardware	2014	Fuchuan et al. [48]
Selection shemes	An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach	1999	Miller [97]
	Cartesian Genetic Programming	2000	Miller and Thompson [100]
	Neutral genetic drift: an investigation using Cartesian Genetic Programming	2015	Turner and Miller [150]
Diversity	Cartesian Genetic Programming	2011	Miller et al. [98]
	Accelerating Convergence in Cartesian Genetic Programming by Using a New Genetic Operator	2013	Meier et al. [93]
Theoretical work	Complexity and Cartesian Genetic Programming	2006	Woodward [165]

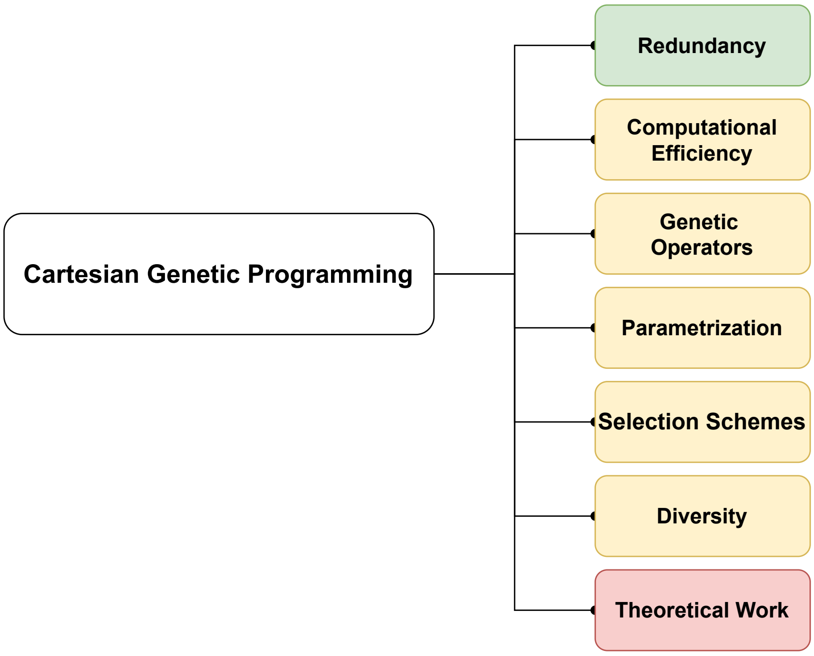


Figure 2.1: The current state of knowledge in the most significant subfields of CGP.

2.2 Thesis Contributions

This thesis presents results of research on fundamental concepts of CGP and introduces advanced techniques for CGP. The objectives of this thesis are twofold on a vertical and a horizontal level. On the one hand the contributions of this thesis contribute vertically on the following fields in CGP: **Genetic operators, Theoretical work, Parametrization, Selection schemes, Diversity**, and **Exploration analysis**. On the other hand, this thesis horizontally contributes to more detailed knowledge about the algorithmic use of CGP and the underlying working mechanisms.

This means that the thesis presents concepts and results that shed more light on the use of CGP, the fundamental working mechanisms, and the generalization of popular dogmas in important subfields of CGP:

- **Theoretical work:** This contribution presents the first theoretical work in CGP. The time complexity of a simple $(1+1)$-CGP algorithm is analyzed with two test problems. These two problems offer different challenges for the CGP standard point mutation mechanism. The lower and upper runtime bounds for both problems are analyzed with the help of *state-of-the-art* methods such as drift analysis and artificial fitness levels.

- **Parametrization:** CGP is mostly used with a parametrization pattern, which has been declared as a general useful pattern for CGP. Furthermore, this pattern has been considered as the most beneficial one for the efficiency of CGP and its variants. This contribution surveys common parametrization patterns with more detailed experiments and reviews alternative patterns. This chapter also analyzes significant CGP literature, which led to the generalization of a parametrization pattern, which is known as (1+4)-CGP. The results of comprehensive experiments lead to new insights and statements about a meaningful parametrization of the CGP algorithm and the functioning of CGP.

- **Diversity:** This contribution describes the first adaptive scheme for CGP, which can significantly contribute to the performance of the real-valued representation. The basic principle of increasing population diversity when the population tends to become too homogeneous is studied on a set of symbolic regression problems. It is also demonstrated that the original introduced real-valued CGP algorithm has a high tendency to stagnate under certain conditions.

- **Genetic operators:** The focus of this contribution is the introduction and investigation of genetic operators, which can significantly improve the convergence behavior and search performance of CGP. Most genetic operators proposed for CGP are functioning on a genotypic level and ignore the functioning of the corresponding phenotype. This contribution introduces new crossover and mutation techniques that work on a phenotypic level. The role of crossover can be seen as a big open question in the field of CGP.

- **Exploration and convergence analysis:** The exploration abilities of CGP algorithms are analyzed in different problem domains. In addition to the analysis of the exploration abilities, the spaces of the fitness values are investigated. This investigation also includes a comparison between the standard point mutation operator of CGP and random initialization of the genotype. The distribution of the fitness values of widespread GP benchmark problems is studied, and the convergence speed is analyzed for different CGP algorithms. Beside this analysis, the convergence speed is measured. Another important contribution of this chapter is an investigation of the behavior of CGP in continuous and discrete fitness spaces.

2.2.1 Research questions

The current state of scientific knowledge in CGP opens some significant research questions:

Research Question 1 (Crossover). *Can some kind of crossover effectively be used in CGP?*

Research Question 2 (Theory). *How can CGP be analyzed on a theoretical level?*

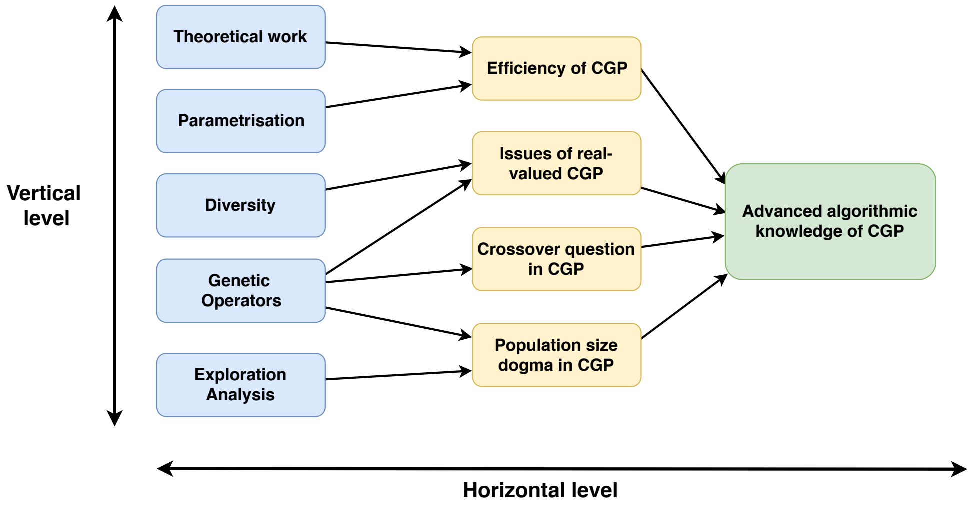


Figure 2.2: Visualization of the contributions of this thesis. The breadth of the tackled points in the thesis is depicted the vertical axis. The interdisciplinary correlations between the respective points are illustrated on the horizontal axis.

Research Question 3 (Real-valued CGP)**.** *Has the real-valued CGP algorithm stagnation problems? Furthermore, if so, why?*

Research Question 4 (Wasted fitness evaluations)**.** *Does the use of middle-sized and big populations in CGP lead to wasted fitness evaluations?*

Research Question 5 (Random Initialization vs. Point Mutation)**.** *Cause random initialization and point mutation similar effects?*

2.2.2 Hypotheses

Based on the current state of fundamental knowledge in CGP and its resulting open research questions, we can formulate the following hypotheses, which will be analyzed in this thesis.

Hypothesis 1 (Population size)**.** *Small populations perform most effective in CGP.*

Hypothesis 2 (($(1 + \lambda)$-CGP)**.** *The $(1 + \lambda)$-CGP algorithm is the most effective way to use CGP.*

Hypothesis 3 (Redundancy)**.** *Extremely large genotypes perform most effectively in CGP.*

Hypothesis 4 (Crossover)**.** *Crossover does not contribute to the search performance of integer-based standard CGP.*

Hypothesis 5 (Mutation)**.** *The standard CGP point mutation operator is sufficient for the mutative variation process.*

Hypothesis 6 (Boolean function domain)**.** *The $(1+\lambda)$-CGP algorithm performs most effectively in the Boolean domain.*

Hypothesis 7 (Symbolic regression domain)**.** *The $(1+\lambda)$-CGP algorithm performs most effectively in the symbolic regression domain.*

2.3 Thesis Organization

The structure of this thesis is as follows.

> **Chapter 3** is devoted to the description of Genetic Programming, Evolutionary Algorithms, Graph-Based Genetic Programming, and CGP. This chapter describes the history of GP and CGP and explains important problem domains relevant for this thesis. This chapter also presents a formal description of CGP.
>
> **Chapter 4** introduces a first runtime analysis for CGP. The work analyzes CGP on a theoretical level using a simple $(1 + 1)$-CGP algorithm. For the analyses of the $(1+1)$-CGP algorithm, two simple and artificial problems are used. The two problems differ significantly in their problem domain and their challenge for the CGP point mutation mechanism. The analysis presents a first analysis of the upper and lower runtime bounds of a $(1+1)$-CGP algorithm.
>
> **Chapter 5** deals with the results and evaluation of experiments on the parametrization of CGP. This chapter mainly focuses on a detailed review of a popular dogma of CGP, which has been generalized in CGP over time. The experiments use the mutation-only CGP, which can be seen as the most used and most popular CGP algorithm. This chapter also outlines and reviews the role of crossover in CGP.
>
> **Chapter 6** is devoted to the diversity problem in real-valued CGP. The real-valued variant of CGP enables the use of an arithmetic crossover technique. However, recent literature outlined that this representation suffers from a decrease in diversity in the population. This chapter investigates the lack of diversity on popular symbolic regression problems and presents an adaptive scheme for real-valued CGP, which contributes to its diversity.
>
> **Chapter 7** introduces two new methods of crossover for CGP. The contribution to the performance of CGP is evaluated in different problem domains and compared to another crossover technique. The chapter also presents a first comparative study of crossover for CGP. The study includes the most popular crossover techniques proposed for CGP and compares the performance of these techniques to the standard $(1 + \lambda)$ CGP algorithm.

Chapter 8 introduces two new mutation operators for CGP. The contribution to the performance of CGP is evaluated in different problem domains. The standard $(1 + \lambda)$ CGP algorithm with different performance patterns is used.

Chapter 9 is devoted to a comprehensive evaluation of the proposed methods. The study compares the fundamental and advanced CGP algorithms. Some of these algorithms are based on the techniques proposed in this thesis. The study also analyzes some algorithms and problems in detail.

Chapter 10 summarizes the contributions and main findings of this thesis. The research questions and corresponding hypotheses are analyzed in this chapter. An outlook illustrates the possibilities for future work on standard CGP, which are based on the contributions of this thesis.

3 Genetic Programming

Genetic Programming can be defined as an evolutionary algorithm-based methodology that enables the automatic derivation of programs for problem-solving. The main idea and motivation behind GP is to evolve a population of candidate *computer programs* toward an algorithmic solution of a predefined problem statement. To accomplish this, GP transforms populations of programs from generation to generation into new populations of programs with (hopefully) higher fitness. GP is an evolutionary algorithm-based methodology. Moreover, GP can be described as an stochastic optimization process, which can consequently not guarantee to achieve the ideal solution. GP counts to the evolutionary computation (EC) domain of the heuristic optimizers family. The first significant work in the field of GP was done by Forsyth [30], Cramer [15], and Hicklin [42]. GP was later significantly popularized by Koza [66, 67, 68]. GP traditionally uses trees as program representation, but it is also used with linear sequence [6, 114, 111], graph-based [121, 144, 97], or grammar-based representations [159, 164, 43, 35]. GP became very popular when Koza applied the syntax tree representation to several types of problems, for instance, symbolic regression, algorithm construction, Boolean function learning, or classification.

GP initializes a population of randomly generated computer programs, which consists of possible functions of the function set. The function set consists of possible functions for the computer programs which are supposed to solve the given problem. GP iteratively transforms a population of computer programs into a new generation by applying the analogs of genetic operations known in the field of evolutionary algorithms (EA) and applied to numerical optimization. These operations are applied to individuals selected from the population. The individuals are selected based on their fitness values. The iterative genetic adaptation of the population toward a predefined goal is performed inside the main generational loop of the run of genetic programming.

GP adapts the principles of familiar evolutionary operators and mechanisms, such as selection, recombination, and mutation. The most popular crossover operator for tree-based GP is the subtree-crossover, which swaps randomly selected sub-branches of two selected trees. The principle of this crossover technique is presented in Figure 3.1. The most popular mutation is named subtree-mutation and generates a random sub-branch, which replaced the sub-branch at a randomly chosen mutation point. The principle of this subtree-mutation is presented in Figure 3.2.

An interesting property of GP is the automatic definition and reuse of subfunctions. Automatically defined functions (ADF) introduce a divide and conquer-like concept to heuristic search. The dynamic extension of the functional block alphabet by new composite functions, which themselves are subject to evolutionary pressure, allows increasing the functionality and complexity level of the alphabet.

GP can be used for a wide range of problems. According to GP survey results [160], trees are the most used representation model in GP. However, since the tree representation does not fit for all types of problems, a graph-based representation is a necessary extension for certain problems. For instance, a syntax tree in standard tree-based GP has only one output. However, for evolving Boolean functions or digital circuits such as digital multipliers or digital adders, a multiple output problem representation is necessary.

The traditional GP model variates candidate programs on a syntactical level. One of the latest introduced GP models, called Geometric Semantic GP, also variates candidate programs on a semantic level.
Besides the main motivation for GP to evolve computer programs, GP has been applied to a diverse set of problems. Some of the most popular problem classes beside to program derivation are symbolic regression, Boolean function learning, classification, and planning. A list of important genetic programming terms, which will be used in the following chapters is given in Table 3.1. A defintion of GP is given in Definition 3.2.

Definition 3.1 (Genetic Programming)**.** *Genetic Programming is an evolutionary algorithm-based method for the automatic derivation of computer programs. Let Θ be a population of $|\Theta|$ individuals and let Ω be the population of the following generation:*

- *Each individual is represented with a genetic program and a fitness value.*

- *Genetic Programming transforms $\Theta \mapsto \Omega$ by the adaptation of selection, recombination and mutation.*

Definition 3.2 (Genetic Program)**.** *A genetic program $\mathcal{P}$ is an element of $\mathcal{T} \times \mathcal{F} \times \mathcal{E}$:*

- *$\mathcal{F}$ is a finite non-empty set of functions*

- *$\mathcal{T}$ is a finite non-empty set of terminals*

- *$\mathcal{E}$ is a finite non-empty set of edges*

Let $\phi : \mathcal{P} \mapsto \Psi$ be a decode function which maps $\mathcal{P}$ to a phenotype Ψ

function set	the set of operators and functions used in a genetic program
terminal set	a set from which all end nodes in the parse trees or input nodes of a graph, representing the programs, must be drawn.
terminal	a variable, constant or a function with no arguments.
non-terminal	functions used to link parse trees together
automatic defined function	a set of sub-trees which can be used as functions in main trees
bloat	phenomenon characterized by the gradual increase of the phenotypes in size during the evolutionary run
goal function	a function (e.g. of mathematical or boolean type) which should be evolved or regressed by the GP algorithm

Table 3.1: List of important terms which are commonly used in the field of Genetic Programming and Cartesian Genetic Programming. These terms and definitions will be used throughout this thesis.

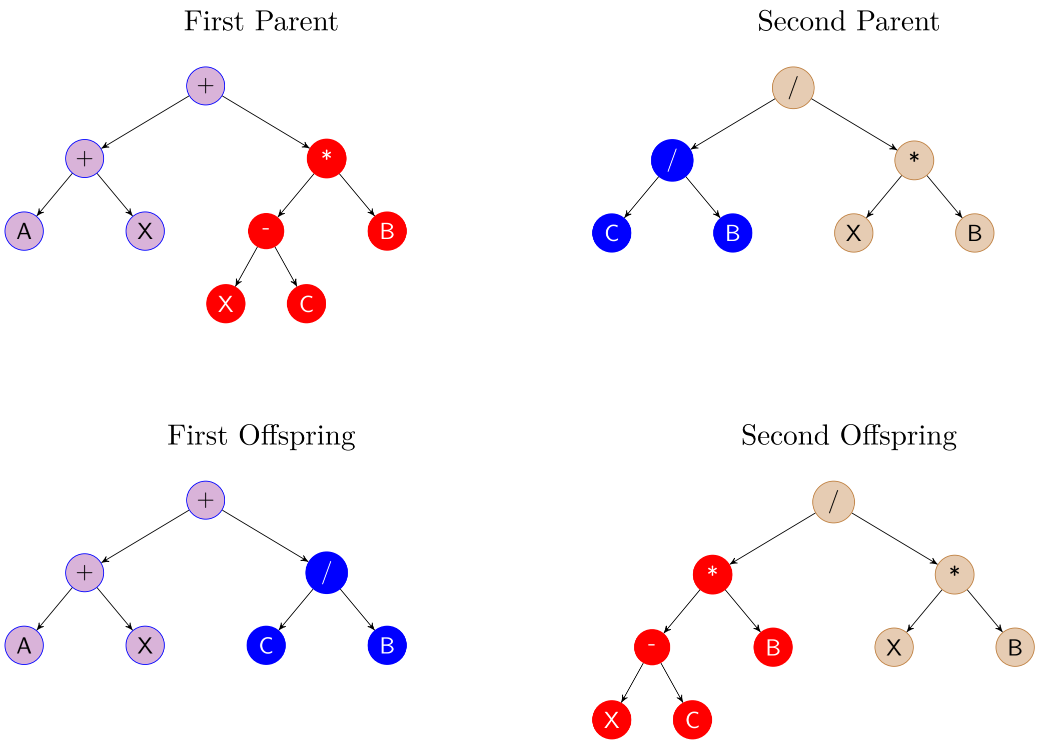


Figure 3.1: Subtree crossover in tree-based GP. The crossover points within of the selected trees are chosen by random. Afterward, the subtrees are swapped and as a result, two offspring are achieved.

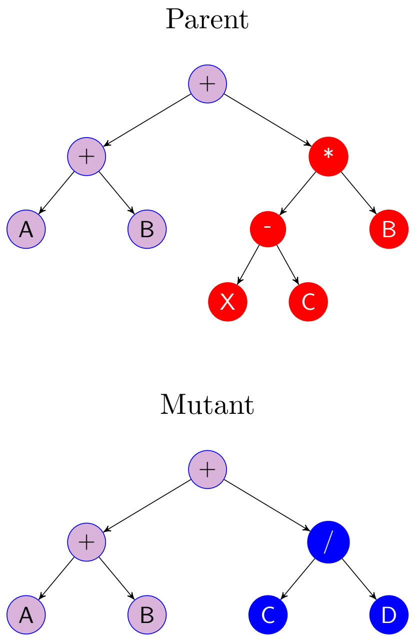


Figure 3.2: Subtree mutation in tree-based GP. A mutation point is chosen by chance and a randomly generated subtree replaces the existing subtree at the mutation point.

3.1 Evolutionary Algorithms

Definition 3.3 (Evolutionary Algorithm)**.** *A metaheuristic optimization algorithm, mostly used for black-box optimization, which adapts the principle of biological evolution.*

Evolutionary Algorithms (EA) adapt the principle of biological evolution, which is commonly referred to as "the survival of the fittest", which was investigated by the English naturalist Charles Darwin in his book *On the Origin of Species by Means of Natural Selection* [16].

Darwin noted with more precise words:

> “As many more individuals of each species are born than can possibly survive; and as, consequently, there is a frequently recurring struggle for existence, it follows that any being, if it vary however slightly in any manner profitable to itself, under the complex and sometimes varying conditions of life, will have a better chance of surviving, and thus be naturally selected.”

Darwin [16, p. 5]

In the field of EC, EAs refer to a group of methodologies which have been introduced in the middle of the 20th century. EA can be considered as metaheuristic optimization algorithms for black-box optimization. Beside the adaption of the main principle of biological evolution, EA mostly adapts two primary genetic operations from nature:

Crossover or recombination
A new possible solution (offspring) is produced from the genetic material of two or more parent individuals.

Mutation
A new possible solution is produced by randomly altering a randomly chosen part of the genetic material of a single individual.

An example of an EA is given in Algorithm 3.1. An EA starts with the initialization of a population with random candidate solutions. A population can be considered as a set of candidate solutions, and each candidate solution owns a genotype. The genotype consists of genes that typically encode a function. A gene can be, for example, a floating-point number, a graph vertex, or a single bit.
In this thesis, we focus on discrete structures such as directed acyclic graphs encoded within a genotype. The encoding and decoding between genotypes and phenotypes are known as *genotype-to-phenotype mapping*. Consequently, a phenotype is the expression or behavior of a genotype. The next step of Algorithm 3.1 is the breeding

population	a set of individuals
species	individuals, which share common characteristics
candidate solution	member of the population, part of the search space
individual	a candidate, potential solution
breeding	the genetic adaption, variation procedure
parent	an individual selected for breeding
offspring	a candidate solution produced by variation
genotype	representation model of an individual, set of genes, vector of numbers
phenotype	expression, behavior of the genotype
chromosome	a set of genotypes
gene	a region of the genotype that encodes functionality
crossover	genetic operator, which combines genetic information of two or more parents to produce new offspring
arithmetic crossover	weighted average recombination (i.e. for real-valued vectors)
one-point crossover	single point recombination (i.e. for discrete bit and real-valued vectors)
mutation	genetic operator, which varies information on the genome of a individual (mostly according to a given probability distribution)
selection	procedure which choses genomes from a population for later breeding
tournament selection	selection strategy that performs multiple tournaments and selects the winners
plus-strategy	parents for breeding are selected from the set of parents and offspring
comma-strategy	parents for breeding are selected only from the set of offspring
fitness function	an objective function to assess and compare individuals by their fitness
fitness	a measurement of the individual's phenotype against the ideal functionality
fitness evaluation	a procedure to evaluate the fitness of each individual
epistasis	expresses the links and interaction between genes in a chromosome and the corresponding effects on the phenotype. Minimal epistasis corresponds to the situation where every gene is independent of all other gene, this is contrary to maximal epistasis where no genes is independent of any other gene.

Table 3.2: List of the important terms which are commonly used in the field of evolutionary algorithms. Some of the listed terms will be used throughout this thesis.

Algorithm 3.1 Example of a simple EA with genetic adaptation, evaluation and selection mechanism

```
procedure Evolutionary Algorithm
    initialize($P$)                               ▷ Initialize set of candidate solutions
    repeat                                        ▷ Until termination criteria not triggered
        $Q \leftarrow$ breed(P)                   ▷ Breed new individuals with crossover and mutation
        Evaluate($Q$)                             ▷ Evaluate the fitness of each individual
        if $Q$ meets termination criterion then
            return $Q$
        end if
        $P \leftarrow$ select(P,Q)                ▷ Select high-fitness individuals
    until $P$ meets termination criterion
    return $P$
end procedure
```

procedure that performs genetic adoption and the exchange of genetic material on the population. More precisely, former selected individuals are recombined and mutated. Thereby, a new individual is created by blending genes from a set of parent individuals. Afterward, a mutation operator modifies some genes with a certain probability. The quality of the newly created individuals is computed in the fitness evaluation step. If at least one individual reaches the optimization goal, the EA terminates its execution. Otherwise, a selection method, which can be deterministic or randomized, picks some individuals regarding their fitness or objective values for the next reproduction step. Table 3.2 provides the explanation of important terms which are frequently used in the field of EAs. A defintion of EA is given in Definition 3.3.

Remarkable historical developments of EAs can be presented and summarized as follows:

1954 •	Barricelli: Evolutionary simulations [106]
1960s - early 1970s •	Rechenberg, Schwefel: Evolutionary strategies [131, 135]
same period of time •	Fogel: Evolutionary programming [29, 27, 28]
same period of time •	Holland: Genetic algorithms [44, 45]
1980s •	Forsysth, Cramer, Hicklin: Genetic programming [30, 15, 42]

Two popular types of EAs are Evolutionary Strategies (ES) [131, 130] and Genetic Algorithms (GA) [46]. An ES operates predominantly in a search space that consists of real values. A vector of real values represents the individuals, and a vector is mutated by adding random numbers. The most popular form of crossover is a weighted average between two real-valued vectors. The weight for the weighted average operation is a random value in the range of 0 and 1. This form of crossover is known as *arithmetic crossover*. The arithmetic crossover is also known as *intermediate recombination* [105]. An ES is described and denoted with the number of parents μ and the number of offspring λ. Another important information is the selection strategy. A *plus-strategy* (denoted by the + operator) selects new parents from the set of offspring and of parents. In contrast, a strategy called *comma-strategy*, which is denoted with a comma, selects parents only from the set of created offspring. The offspring are selected by the rating of their fitness value. If only one parent is used, which creates one offspring, and the new parent is selected from the set of the offspring and the current parent, the ES is denoted as $(1+1)$-ES. If the ES creates λ offspring from a single parent in each generation, the ES is called $(1+\lambda)$-ES.

In contrast, a typical GA operates in a search space of discrete bit values. More precisely, GAs typically represent individuals as a vector of bit values. Each new offspring is produced by combining parts of the bit vector from each parent. This is analogous to the way chromosomes of DNA, which contains the inherited genetic material, are passed to children in natural systems. One popular method of recombining the genetic material is called *one-point crossover*. This method chooses a random point in the vector and transfers the front part from the first parent to the offspring. The rear part is then transferred from the second parent to the offspring. GAs are often used with a generational EA and a selection method called *tournament selection*. Tournament selection is a strategy to select parents from the selection pool where their fitness value compares a predefined number of randomly selected individuals. These sets of randomly selected parents are called tournaments. The individual with the best fitness value wins the tournament. A big advantage of this selection method is that the size of the tournament can control the selection pressure of the EA. The evolutionary procedure, which is shown in Figure 3.4 refers to the GA, which has been proposed by Holland and is known as *Canonical Genetic Algorithm*.

GP can be seen as one outcome and derivation of the development of GAs. Especially conventional tree-based GP is mostly used with a basic GA. However, CGP is also used with an algorithmic pattern which is derived from ESs. CGP can be seen as a derivation from GP. The visual taxonomy of the development toward GP and CGP is given in Figure 3.3.

3.2 Graph-based Genetic Programming

One of the first graph-based GP approaches was introduced by Poli [121, 120, 122, 123, 124] called parallel distributed Genetic Programming (PDGP), a form of GP for the evolution of parallel programs, which reuses partial results. PDGP uses a rectangular or irrectangular grid representation. In PDGP, the graphs are represented with nodes that stand for functions and terminals. Each node occupies a position in the grid.

The edges of the PDGP representation stand for the control of the data flow. The genetic operators of PDGP are crossover and mutation.

PDGP uses a direct representation of graphs. The representation is based on the

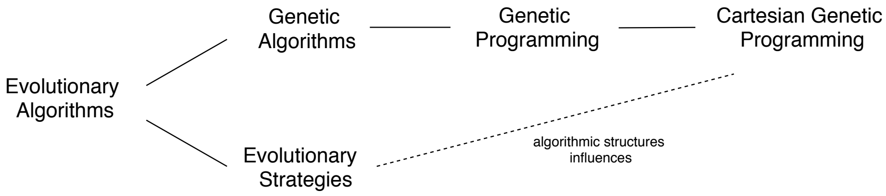


Figure 3.3: Taxonomy of some popular evolutionary algorithms. Since the first approaches to GP used genetic algorithms, the field of GP grew out of the developments, which occurred in the field of genetic algorithms. CGP can be seen as an extension of GP. However, since CGP also uses algorithmic selection schemes which are used to run evolutionary strategies, CGP was also influenced by this type of evolutionary algorithms.

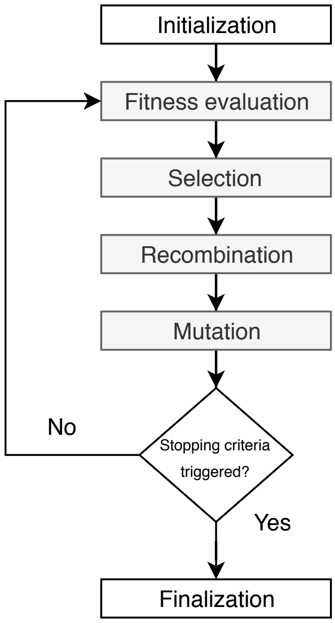
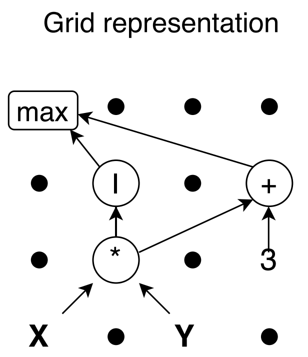


Figure 3.4: Flowchart of a typical canonical genetic algorithm. Recombination and mutation are mostly used for the breeding procedure. Among genetic algorithms, tournament selection can be seen as the most popular method for the selection process.



List representation

max	(0,0)	+1	+3
I	(0,1)	0	
+	(3,1)	-2	0
*	(1,2)	-1	+1
3	(3,2)		
X	(0,3)		
Y	(2,3)		

Figure 3.5: Program representation in PDGP for the program: A visual representation is achieved with grids and a syntactic representation is achieved with lists. The listing contains the label, the coordinates of the node, and the horizontal displacement of the nodes in the previous layer whose value is used as an argument for the node.

idea of assigning each node in the graph to a location in a multi-dimensional and evenly spaced grid with a regular or irregular shape and limiting the connections between nodes and be upwards. The PDGP representation allows connections to exist only between nodes belonging to adjacent rows. The representation for parallel distributed programs is illustrated by an example program in Figure 3.5, where the program has a single output at coordinates (0,0) and the y axis is pointing downwards. Pass-through nodes can be constructed by adding the identity function to the function set. In this way, any parallel distributed program can be rearranged in a way that it can be described with the grid-based graph representation of PDGP. Besides the grid-based representation it is possible to describe any program by listing of the following parameters for each node: 1) the label, 2) the coordinates of the node, and 3) the horizontal displacement of the nodes in the previous layer whose value is used as argument for the node.

Another graph-based GP approach called Parallel Algorithm Discovery and Orchestration (PADO), which has been proposed by Teller [144], uses a combination of GP and linear discrimination to obtain parallel classification programs. PADO programs include three parts: the main loop, some automatically defined functions (ADFs), and an indexed memory. The main loop is repeatedly executed for a fixed amount of time. When the time is up, PADO programs are forced to halt by some external control structure. The output of a program is the weighted average of the outputs produced at each iteration of the loop. The weights are proportional to the iteration count so that more recent outputs count more.

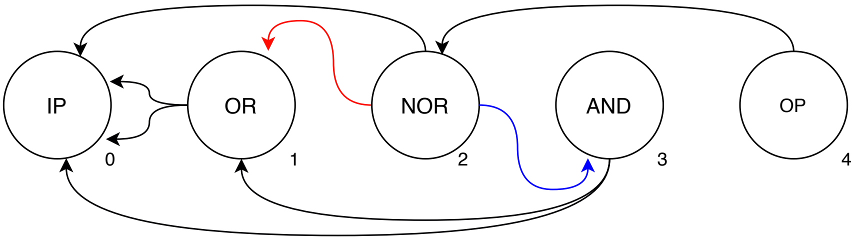


Figure 3.6: Examplification of an allowable graph transformation of a Boolean function which is not allowed in CGP. An EGGP mutation which changes a connection (red) from node 2 to node 1 is replaced with a connection (blue) directed to node 3. This mutation produces a valid circuit but is impossible in standard CGP as mutations on the connection genes have values less than the node position which is done to guarantee feedforward property.

The most popular and dominant graph-based GP approach is CGP, which was finally introduced by Miller [97, 100], will be described in the next section. Compared to the other graph-based GP approaches, CGP can be considered as the most used graph-based GP approach, which can be applied to a wide range of problems.
A comparatively new method for evolving graphs has been proposed by Atkinson et al. [3, 4, 2]. It is called evolving graphs by graph programming (EGGP) and evolves graphs directly instead of using a grid-based genotype. Basically, EGGP utilzes a probabilistic extension for the graph programming language GP2 [119, 118]. GP2 itself is a rule-based, nondeterministic programming language for solving graph problems at a high level of abstraction. In EGGP, some mutations are not possible in the standard form of CGP due to the feed-forward preservation. Figure 3.6 shows an allowable mutation in EGGP which is not allowed in standard CGP.

3.3 Cartesian Genetic Programming

CGP is a form of Genetic Programming, which offers a graph-based representation. In contrast to tree-based GP, CGP represents a genetic program via genotype-phenotype mapping as an indexed, acyclic, and directed graph. Originally, the structure of the graphs was a rectangular grid of n_r rows and n_c columns, but later work focused on a representation with one row. The CGP decoding procedudure processes groups of genes, and each group refers to a node of the graph, except the last one, which represents the outputs of the phenotype. Each node is represented by two types of genes which index the function number in the GP function set and the node inputs. These nodes are called *function nodes* and execute functions on the input values. The number of input genes depends on the maximum arity n_a of the function set.
Given to the number of outputs n_o, the n_o last genes in the genotype represent the indexes of the nodes, which lead to the outputs. A backward search is used to

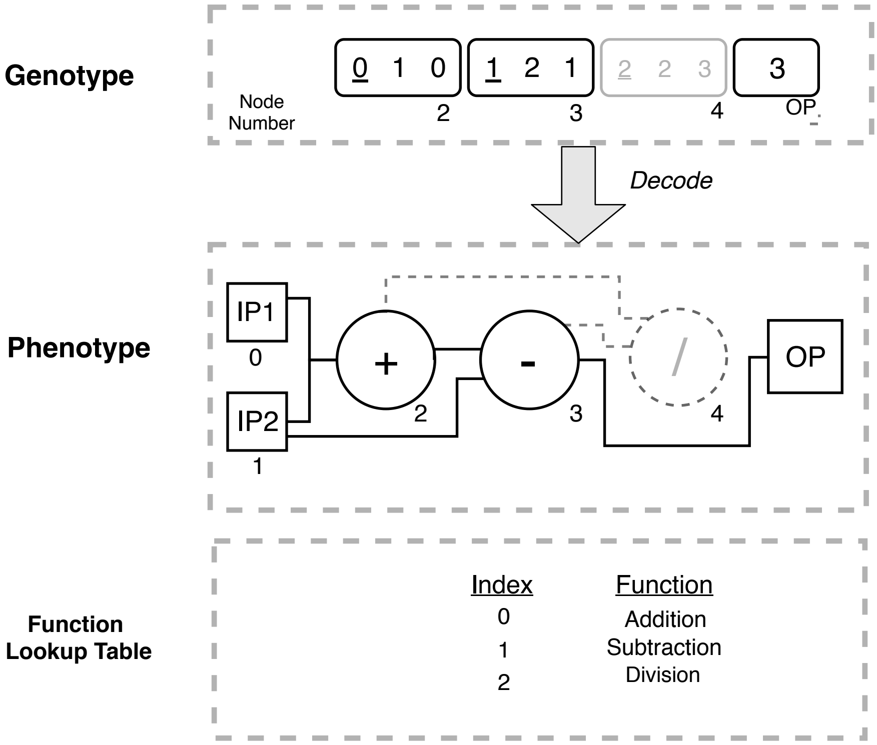
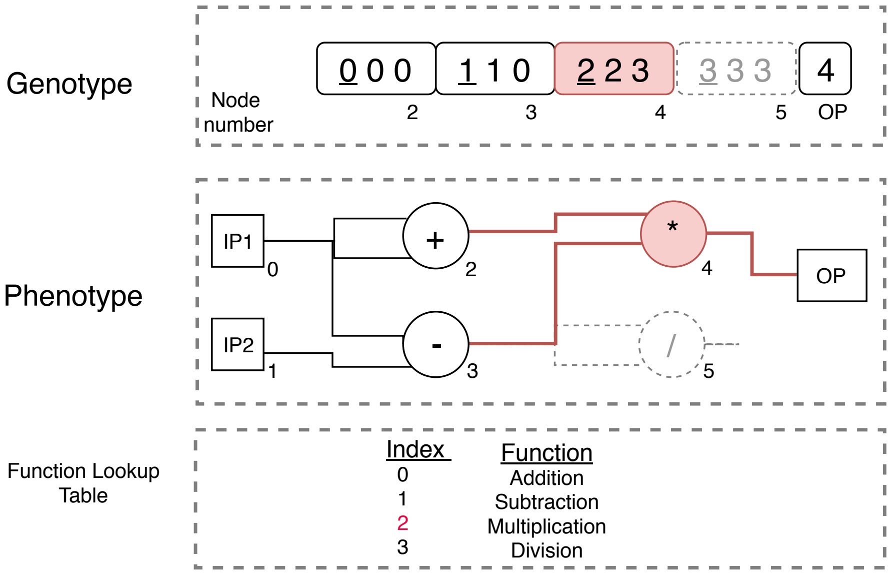


Figure 3.7: The exemplification of the decoding procedure of a CGP genotype to its corresponding phenotype. The nodes are represented with two types of numbers which index the number in the function lookup table (underlined) and the inputs (non-underlined) for the node. Inactive function nodes are shown in gray. The identifier IP1 and IP2 stand for the two input nodes with node index 0 and 1. The identifier OP stands for the output node of the graph.

decode the corresponding phenotype. An example of the backward search of the most popular one-row integer representation is shown in Figure 3.7. The backward search starts from the program output and processes all nodes which are linked in the genotype. In this way, only active nodes are processed during evaluation. The genotype in Figure 3.7 is grouped by its function nodes. The first (underlined) gene of each group refers to the function number in the corresponding function set in the figure. The decoding procedure of one function nodes is shown in Figure 3.8. The decoded part of the genotype and its corresponding part in the phenotype are highlighted in orange. The integer-based representation of CGP phenotypes is mostly used with mutation only.

Early studies on efficiency showed that several genetic crossover operators do not contribute to the search performance of CGP. The number of inputs n_i, outputs n_o, and the length of the genotype is fixed. Every candidate program is represented with $n_\mathrm{r} * n_\mathrm{c} * (n_\mathrm{a} + 1) + n_\mathrm{o}$ integers. Even when the length of the genotype is fixed for each



Index	Function
0	Addition
1	Subtraction
2	Multiplication
3	Division

Figure 3.8: Decoding procedure from genotype to phenotype illustrated for one function node with node number 4 (highlighted in colour). The function of a certain function node is represented with the function gene. In the figure, the function gene is underlined. The input connections from the outputs of former function nodes are represented by the input genes which follow the function gene in the genotype.

candidate program, the length of the corresponding phenotype in CGP is variable, which can be considered as a advantage of the CGP representation.
CGP is traditionally used with a (1+λ) selection scheme of EAs. A pseudo-code is given in Algorithm 3.2. The new population in each generation consists of the best individual of the previous population and the λ created offspring. The breeding procedure is mostly done by a point mutation that swaps genes in the genotype of an individual in the valid range by chance. An example of a point mutation is given in Figure 3.9. The figure shows the flip of the value of a connection gene, which rewires the corresponding function node. Another point mutation is the flip of the functional gene, which changes functional behavior of the corresponding function node.

Algorithm 3.2 (1+λ)-EA

```
initialize($P$)                                                    ▷ Initialize parent individual
repeat
    $Q \leftarrow$ breed(P)                                        ▷ Breed $\lambda$ offspring by mutation
    Evaluate($Q$)                                                  ▷ Evaluate the fitness of the offspring
    if at least one individual of $Q$ has better fitness then $P$ then
        $P \leftarrow$ best(Q)                                     ▷ Replace the parent by the best offspring
    end if
until $P$ meets termination criterion                              ▷ Until termination criteria not triggered
return $P$
```

The (1+λ)-CGP is often used with a selection strategy called *neutrality*, the idea that genetic drift yields to diverse individuals having equal fitness. The genetic drift is implemented into the selection mechanism in a way that individuals that have the same fitness as the normally selected parent are determined, and one of these same-fitness individuals is returned uniformly at random. The connectivity of the graph can be controlled with the parameter l called *levels-back* which constrains from which previous columns a function node can receive its input connections.

Some pivotal advantages of CGP are:

- The maximal size of encoded solutions is bounded, saving CGP to some extent from "bloat" that is characteristic to GP [75, 71, 140, 92].
- CGP encodes a directed acyclic graph (DAG), which allows the evolution of structures which can be represented as DAGs. In this way, CGP also facilitates to evolve topologies. An example is the evolution of artificial neural network using CGP [149].
- CGP offers an implicit way of propagating redundant information throughout the generations. This is accomplished using extremely high levels of redundancy where over 95% of the genes in the CGP genotype are inactive. Since

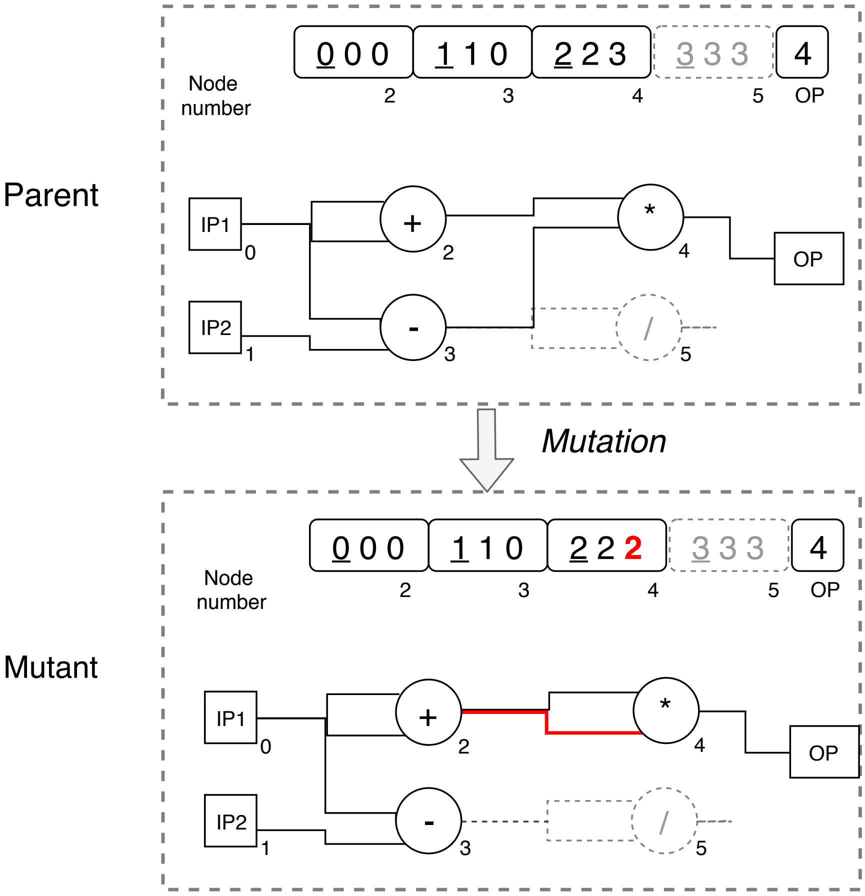


Figure 3.9: Exemplification of the standard point mutation operator in integer-encoded CGP. Genes of the genotype are selected by chance, and their values are randomly flipped within the legal range of possible values. The connection gene of node 4 is mutated from a value of 2 to a value of 3. This causes rewiring of the second input of node 4 from the output of node 3 to the output node of node 2.

many genes in CGP are redundant, these genes can be used as a source of randomness and memory for evolutionary artifacts that have no effect on the phenotype. Redundant genes in CGP assist in effective evolutionary search [99].

- The implementation complexity of CGP is low, where no special programming language properties are necessary, in contrast to tree-based GP where the ability to handle tree structures efficiently is crucial.

Along with the mentioned advantages, CGP suffers as a combinatorial representation model from the usual sources of epistasis. For instance, rewiring a single input of a functional node can change the phenotype dramatically. Additionally, the spatial arrangement of functional nodes on a two-dimensional grid introduces restrictions on the topology of the evolved solutions. Moving a function node among the grid requires rearranging the genotype, if possible. Additionally, the connection settings of the input of a function node strongly depend on the location of the node on the grid. These dependencies implicitly have impact on the evolvability and make it challenging to realize structural methods for CGP. A trial to free CGP from grid-induced epistasis was made in [139] by assigning a signature to each input and output of a node. Best-fitting signatures were used to clamp wires and in this way the mapping between the genotype and phenotype is determined by self-organized binding of the genes which is inspired by enzyme biology.

In contrast to tree-based GP, CGP is used primarily with mutation as the sole genetic operator. The reason for this is that standard genotypic operators failed to improve the search performance of standard CGP. A defintion of CGP is given in Definition 3.4. The described functionality relates to standard CGP. Later work enabled several variants of CGP, which will be surveyed in the historical overview.

Definition 3.4 (Cartesian Genetic Programming)**.** *Cartesian Genetic Programming is a form of Genetic Programming, which offers a graph-based representation model for Genetic Programming based on a rectangular grid or row of nodes.*

3.3.1 Formal description of CGP

For the formal definition of CGP we first formally define a *Cartesian Genetic Program* (CP) in Definition 3.5:

Definition 3.5 (Cartesian Genetic Program)**.** *A cartesian genetic program $\mathcal{P}$ is an element of $\mathcal{N}_{\mathrm{i}} \times \mathcal{N}_{\mathrm{f}} \times \mathcal{N}_{\mathrm{o}} \times \mathcal{F}$:*

- *$\mathcal{N}_{\mathrm{i}}$ is a finite non-empty set of input nodes*

- *$\mathcal{N}_{\mathrm{f}}$ is a finite set of function nodes*

- $\mathcal{N}_\mathrm{o}$ *is a finite non-empty set of output nodes*

- $\mathcal{F}$ *is a finite non-empty set of functions*

All nodes of a CP are continuously indexed with a node number. The indexing starts with the a value of 0 at the first input node and ends at the last output node. At each node, the node number is increased by one. Let $N = |N_\mathrm{i}| + |N_\mathrm{f}| + |N_\mathrm{o}|$ be the number of nodes of a CP.

Given the number of input nodes n_i, the set $N_\mathrm{i} = \{\theta_i^j \mid j \in \{0, \ldots, n_\mathrm{i}-1\}\}$ consists of n_i input nodes. An input node θ_i can be formally described as a 2-tuple $\theta_\mathrm{i} = (x_i, v_i)$:

- $x_i \in \{0, \ldots, n_\mathrm{i} - 1\}$ is the number of the input node
- v_i is the value of the input node

Given the number of function nodes n_f, the set $N_\mathrm{f} = \{\theta_f^j \mid j \in \{0, \ldots, n_\mathrm{f} - 1\}\}$, consists of n_f function nodes. Given the maximum arity n_a, a function node θ_f can be formally described as a tuple $\theta_\mathrm{f} = (x_f, g_f, g_c^0, ..., g_c^{n_a-1})$ of dimension $n_\mathrm{a} + 2$:

- $x_f \in \{n_\mathrm{i}, \ldots, n_\mathrm{i} + n_\mathrm{f} - 1\}$ is the number of the function node
- $g_f \in \{x \in \mathbb{N}^0 \mid 0 \leq x \leq |F| - 1\}$ is the function gene
- $g_c^0, ..., g_c^{n_a-1} \in \{x \in \mathbb{N}^0 \mid 0 \leq x \leq x_\mathrm{f} - 1\}$ are the connection genes

Given the number of output nodes n_o,the set $N_\mathrm{o} = \{\theta_o^j \mid j \in \{0, \ldots, n_\mathrm{f} - 1\}\}$, consists of n_o output nodes. An output node θ_o can be formally described as a 2-tuple $\theta_o = (x_o, g_\mathrm{o})$:

- $x_o \in \{n_\mathrm{f}, \ldots, N - 1\}\}$ is the number of the output node
- $g_o \in \{x \in \mathbb{N}^0 \mid 0 \leq x \leq |G_\mathrm{i}| + |G_\mathrm{f}| - 1\}$ is the output gene

An item ρ of the function set F can be described as a 3-tuple $\rho = \{i_\rho, f_\rho, a_\rho\}$:

- $i_\rho \in \{x \in \mathbb{N} \mid 0 \leq x \leq |F| - 1\}$ is the index

- f_ρ is the function

- $a_\rho \in \mathbb{N}$ is the arity of f_ρ

The output gene g_o of each output node contains the node number of a function or input node whose node number is smaller than the node number of the output node. The output gene is used as a reference when the graph is decoded. When the output gene references to a function node, the respective function node is decoded in two ways: (1) The function gene g_f of the function node θ_{f} is used to decode the function f_ρ in the function set F which is executed by the node. The function gene contains the index i_ρ of an item ρ in the function set F. (2) The connection genes $g_c^0, ..., g_c^{n_a-1}$ represent inputs of the node which are connected with the outputs of previous function and input nodes. The values of the connection genes must be smaller than the node number of the respective function node to ensure the feed forward structure of the graph after decoding. The decoding itself starts at the output nodes and continues until the inputs nodes are reached. The decoding procedure is done for all output genes. The result of the decoding procedure can be described as a set of directed paths Ω. Given the input set I and the output set O, let $\omega = I \times \Omega \mapsto O$ be an output function.

Table 3.3: Identifiers for various CGP algorithms

Identifier	**Description**
$(1+\lambda)$-CGP	$(1+\lambda$ selection strategy with neutral genetic drift
$(\mu+\lambda)$-CGP	$(\mu+\lambda)$ selection strategy without neutral genetic drift
Canonical-CGP	Canonical EA with tournament selection
Pop50-CGP	Canonical EA with a population size of 50 individuals
Real-valued-CGP	Canonical EA with real-valued representation

3.3.2 Real-valued Representation of CGP

To enable the use of crossover in CGP, Clegg et al. [13] introduced a real-valued representation for CGP. The real-valued representation of the CGP genotype, as shown in Figure 3.10 enables the recombination of two genotypes by an arithmetic crossover technique. The values of the genes vary in the interval [0,1].

The genotype containing $\mathcal{P}$ genes of an real-valued CGP individual is initialized with a tuple $G = \{x_p\}_{p=1}^{\mathcal{P}}, x_p \in U[0,1]$ set of $\mathcal{P}$ continuous uniform distributed random points.

The genes are decoded to the integer-based representation by using normalization values (e.g. the number of functions or maximum input range) as shown in Figure 3.10. The decoding procedure is also illustrated in Figure 3.10. The mutation on this representation is done by replacing the gene value with a decimal random value which varies in the interval [0,1]. For mutation, certain genes in the genotype are selected with respect to a predefined mutation rate and the gene values are altered by chance in the interval [0,1]. Clegg et al. outlined that the real-valued and integer-based representations show similar convergence behavior when only mutation is used as the sole genetic operator.

Furthermore, Clegg et al. concluded that the new representation, along with crossover improves the convergence behavior for the first generations. For the latter generations, the use of crossover in real-valued CGP was found as not beneficial for the search performance on one of the two tested symbolic regression problems. Later, work [55] demonstrated that the efficiency of real-valued CGP can be improved by maintaining population diversity.

Some identifiers for EAs and certain selection schemes, used with CGP, are oriented with identifiers which are known in the CGP community and have been used in CGP literature. The identifiers which are used in this thesis are listed in Table 3.3.

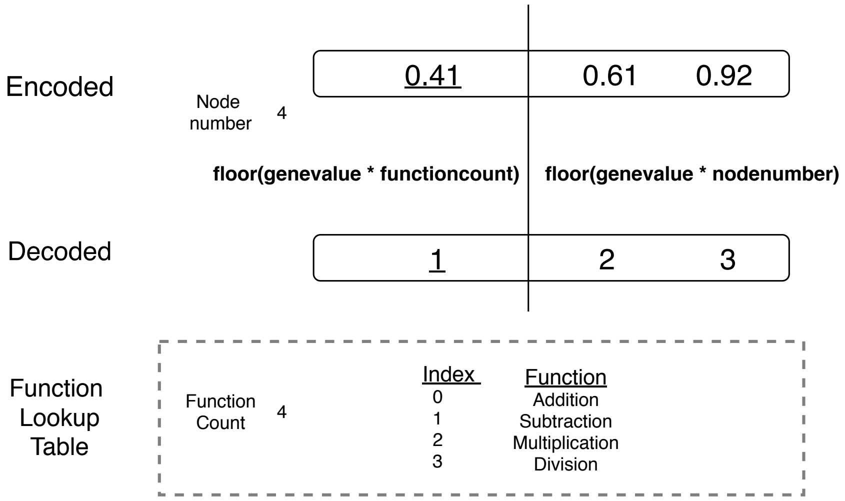


Figure 3.10: Exemplification of the decoding procedure from real-valued to the integer-based genotype. In real-valued CGP, the values of all genes vary in the interval [0,1] and are decoded to a non-negative integer number. The function gene is decoded by its value and the number of functions in the function set. This is done by multiplying the value of the function gene with the number of functions in the function set. In this example, we find four functions in the function set. The input genes are decoded by the node number of the respective function node. The decoding is also done with a multiplication, in this case between the node number and the value of the connection gene. After the multiplications for the function and connection gene, the floor function is applied to the result of the multiplications.

3.4 Historical Overview of GP and CGP

Forsyth [30], Cramer [15] and Hicklin [42] are considered to have published the earliest work on GP. Forsysth introduced BEAGLE, which evolves programs using an algorithm that is similar to basic GA. The rules are tree-structured Boolean expressions, including arithmetic operations and comparisons. The rules of BEAGLE use their own language, but Forsyth did also suggest that at some point in the future, LISP might also be suitable. In his work, Cramer presented a representation for the adaptive generation of simple sequential programs. Hicklin used a genetic algorithm for the automatic program generation. The created functions were written in the simple "number-string" computer language JB [15], and in TB [15], a modified version of JB with a tree-like structure. Cramer discussed the use of an adaptive GA to permit the adaptive generation of simple computer functions from low-level computational primitives. Koza [66, 67, 68] can be considered as one of the main establishers of the field of Genetic Programming and significantly popularized GP in the early 90s. Koza's work on a syntax tree representation model for the programming language LISP outlined the powerful abilities of GP in different problem domains. Furthermore, Koza presented first ideas applied on a grammar-based variant of GP, which led to the introduction of grammar-based Genetic Programming by Wong and Leung [163]. With the intention to evolve modular and hierarchical structures, Koza introduced automatically defined functions (ADFs) [68] which are considered to be the most widely used method of evolving reusable components. Basically, ADFs use a fixed architecture, which are specified in advance by the user. Koza later extended this using architecture-altering operations which allow the architecture to evolve along with the programs. The use of a stepwise adaptation of weights technique (SAW) applied to GP led to the development of *Adaptive GP* [23, 24]. The SAW mechanism has been originally developed for and successfully used in EAs for constraint satisfaction problems [25, 26, 5]. Other representations of GP such as graph-based [144, 121, 100] and linear-based GP [7] (LGP) can be considered because of Koza's effort. A further development on linear-based GP was made with the introduction linear-graph GP [58], which has been developed with the goal of giving a program the exibility to choose different execution paths for different inputs. This should enable the evolution of programs of higher complexity, that can compete with the complexity and possibilities of hand-written programs. Autoconstructive evolution was illustrated with Pushpop, an approach of evolving population of programs which are expressed in the Push programming language [141, 142]. A new approach to program representation was made with the introduction of Enzyme GP [82, 79, 81, 80] where interactions between program components are expressed in terms of a component's behavior and not through its relative position within a representation. Geometric Semantic Genetic Programming (GSGP) as introduced by Moraglio [102] can be considered as one of the most recent works and development for tree-based GP. Motivated by the issue of code disruption in LGP, a new form of LGP called Parallel LGP (PLGP) was introduced [22]. PLGP programs consist of lists of instructions which are executed in parallel. The resulting vectors are then combined to produce the output of the program. PGLP limits the disruptive effects of crossover and mutation, which allows PLGP to significantly outperform regular LGP. An interesting and practical subfield of GP is Genetic Improvement (GI), which means the use of optimization and machine learning techniques, such as GP, to improve existing the code of software [73, 115].

The timeline of GP can be represented as follows:

1981 •	Forsysth: BEAGLE
1985 •	Cramer: Adaptive Generation of Simple Sequential Programs
1986 •	Hicklin: Application of the genetic algorithm to automatic program generation
1990 •	Koza: Genetic Programming on LISP Syntax Trees
1990 •	Koza: First Ideas on Grammar Based Genetic Programming
1993 •	Banzhaf: Linear Genetic Programming
1994 •	Koza: Modular Genetic Programming
1995 •	Wong, Leung: Grammar Based Genetic Programming
1996 •	Poli: Parallel Distributed Genetic Programming
1996 •	Teller: Parallel Algorithm Discovery and Orchestration
1997 •	Miller, Thompson, Kalganova, Fogarty: Steps forward Cartesian Genetic Programming
1999 •	Miller: Cartesian Genetic Programming
1999 •	Eggermont, Eiben, van Hemert: Adaptive GP
2001 •	Spector: PushGP
2002 •	Kantschik, Banzhaf: Linear-Graph GP
2003 •	Lones and Tyrell : Enzyme Genetic Programming
2011 •	Parallel Linear Genetic Programming
2011 •	Moraglio: Geometric Semantic Genetic Programming
2015 •	Langdon: Genetic Improvement
2018 •	Atkinson, Plump, Stepney: Evolving Graphs by Graph Programming

Related to CGP, the historical overview starts with the earliest work published by Miller, Thompson, Kalganova, and Fogarty, which included the first proposals for the standard CGP encoding model inspired by the two-dimensional array of functional nodes connected by feed-forward wires of an FPGA device [95, 50]. Miller [97, 100] finally introduced CGP and demonstrated its abilities to solve Boolean function problems. A first extension to the standard CGP model was made with the introduction of Modular CGP by Walker et al. [155], which enables the use of ADFs in CGP. Embedded CGP, as introduced by Walker et al. [157] is an extension of CGP, which is capable of automatically acquiring, evolving, and re-using partial solutions in the form of modules. With the intention to use a representation in which the specific location of genes within the chromosome has no direct or indirect influence on the phenotype, Smith et al. [139] introduced an implicit context representation for CGP. Self-Modifying-CGP was developed by Harding et al. [36]. It uses functions that cause the evolved programs to change themselves as a function of time. By using this technique, it is possible to find general solutions to classes of problems and mathematical algorithms (e.g. arbitrary parity, n-bit binary addition, sequences that provably compute π and e to arbitrary precision). An approach to CGP encoded artificial neural networks, which was introduced by Turner et al. [148] offers a powerful training method for neural networks. This is because CGP can simultaneously evolve the network connection weights, topology, and neuron transfer functions. It is also compatible with Recurrent-CGP, which enables the evolution of recurrent neural networks. Recurrent-CGP, as introduced by Turner et al. [146, 151] allows evolution to create programs that contain cyclic, as well as acyclic, connections. This enables the application to tasks that require internal states or memory. It also allows CGP to create recursive equations.

Iterative CGP, as introduced by Ryser-Welch [132] enables the automatic creation of human-readable algorithms. In Positional CGP [162] (PCGP), node positions are evolved and nodes can be added or removed from a genome without disturbing the existing connection scheme, unlike in CGP, where a node addition and deletion cause a shift in all downstream node positions.

Reviewing the most significant literature of CGP, the timeline of CGP can be represented as follows:

1999	Miller, Thompson: Standard CGP
2004	Walker, Miller: Modular CGP
2005	Smith, Leggett, Tyrrell: Implicit Context Representation of CGP
2007	Clegg, Walker, Miller: Real-valued representation of CGP
2008	Walker, Miller: Embedded CGP
2011	Harding, Miller, Banzhaf: Self-Modifying CGP
2011	Harding, Graziano, Leitner, Schmidhuber: Multi-type CGP
2013	Turner, Miller: CGP encoded artificial neural networks
2014	Turner, Miller: Recurrent CGP
2015	Kalkreuth, Krone, Rudolph: Adaptive real-valued CGP
2016	Ryser-Welch, Miller, Swan and Trefzer: Iterative CGP
2018	Wilson, Miller, Cussat-Blanc, Luga: Positional CGP

3.5 Problem Domains of GP and CGP

To describe some of the most prevalent problem domains of GP and CGP of the present and past, we survey three problem domains in this section. However, the set of possible application areas of GP is diverse. As a compromise, this section focuses on areas where GP and CGP have been successfully applied and that are most relevant for the experiments of this thesis.

3.5.1 Symbolic Regression

Symbolic regression is one of the most popular problem domains in GP. Symbolic regression problems can easily be represented as syntax trees, and in this way, these problems can be solved using evolutionary adaptation. In the first place, the symbolic regression domain is often used for the tuning of new algorithms. It is also widely used in real-world applications, where other regression methods are unsuccessful. Symbolic regression problems are often tackled with a set of arbitrary data. The challenge for GP is to find a solution (e.g. mathematical expression) that fits the data points of the given regression function. An example for a mathematical expression, which is represented as a tree is given in Figure 3.11. Usually, a cost function with a measure like root mean square error or the absolute difference between the data points of a possible solution and the actual values of the regression function is used to evaluate the fitness of the individuals.

A symbolic regression problem in GP can be formally described with the following definition:

Given a training dataset $T = \{x_p\}_{p=1}^{\mathcal{P}}$ of $\mathcal{P}$ random points, a function $f_{\text{ind}}(x_p)$ that returns the value of an evaluated individual and a functions $f_{\text{ref}}(x_p)$ which returns the true function value, let

$$C := \sum_{p=1}^{\mathcal{P}} |f_{\text{ind}}(x_p) - f_{\text{ref}}(x_p)|$$

be the cost function, which calculates the sum of absolute differences between the value of the evaluated individual and the true function value.
In most cases, after a solution was found which fits the given training dataset, the generalization abilities of the solution are evaluated with an independent validation dataset.

Basically, symbolic regression is not restricted to numerical data sets. The induction of compact Boolean expressions based on truth tables can also be applied with GP. For these types of problems, independent and dependent variables are of Boolean type.

Symbolic regression, along with GP, got popular through Koza's experiments with four, five, and six order polynomial functions. Today, these types of problems are of insignificant meaning in terms of practical application in GP and are mostly referred to as *toy problems*. However, for the analysis of GP algorithms, these problems are significant and can lead to more profound insight into the specific behavior of GP algorithms and techniques.

3.5.2 Boolean Functions

A range of Boolean functions can be composed of GP syntax trees. The terminals (leaves) of the trees represent the Boolean inputs. This problem domain has been mainly used for benchmarking GP algorithms and techniques by evolving Boolean single output problems such as multiplexers or parity-even functions. CGP has been successfully applied to multiple-output problems such as multiplier, adders, or subtractors. The challenge for GP is to fit a truth table for the given Boolean function.

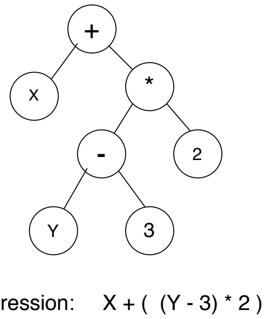


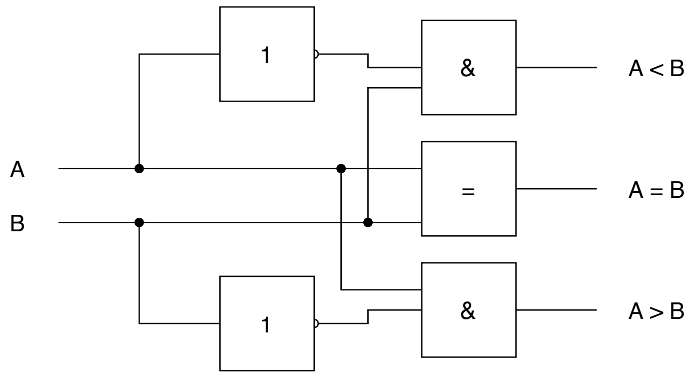
Figure 3.11: The mathematical expression $X+((Y-3)\cdot 2)$ represented as a parse tree as it is used in tree based GP. The mathematical expression is decoded by starting at the leafs, which represent the terminals and constants. In this case X, Y are terminals and 3 and 2 are constants. The leafs of a node are the arguments for the respective mathematical functions.

The number of different bits represents the fitness. Another way how fitness is represented in the Boolean domain is the number of parity-even or parity-odd bits of the given problem. The high interest of CGP's application in the Boolean function domain turned out from the motivation for developing CGP. Since directed and acyclic graphs can have multiple outputs, CGP can easily be applied to the design of digital circuits, which gates consist of Boolean functions such as AND, OR, NAND or XOR. Furthermore, since the genotype of the CGP representation is fixed and can be set to a maximum number of function nodes, CGP can easily evolve digital circuits or other compositions of Boolean functions, which must be limited to a maximum number of nodes.

3.5.3 Image operator design

Image operator design problems can be considered as a typical case where a graph representation can be beneficial. For the design of specific image operators, for instance, operators of FPGA devices, a matrix of pixel values from a particular image is used for the input of the genetic program. Since a graph representation offers connections from one specific function node to any previous node of the graph, this behavior can be used for the automatic design of image operators. Image operators also require flexible connections between function and input nodes, which can easily be handled by a graph representation. Furthermore, the graph representation can also be used for the design of operators with multiple outputs.

An example is the design of an edge detector for which the detection of edges in horizontal and vertical direction is significant. The function set for these types of problems typically exists of low-level image processing operations such as logical and arithmetic functions. The fitness functions for this type of problem are often defined as the mean difference between the pixel values of the input image and the pixel values of a reference image. An example for the use of this fitness function is the



Truth Table

A	B	A < B	A = B	A > B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

Figure 3.12: An exemplification 1-Bit boolean comperator which can be evolved with represented with the CGP representation model. The circuit has two boolean inputs A and B and three boolean outputs which represent the boolean conditions $A < B$, A = B and $A > B$. The corresponding truth table is used to evaluate the fitness of candicate solutions. The fitness function is often defined as the number of similar bits between the output bits of the candidate solution and the output bits of the truth table.

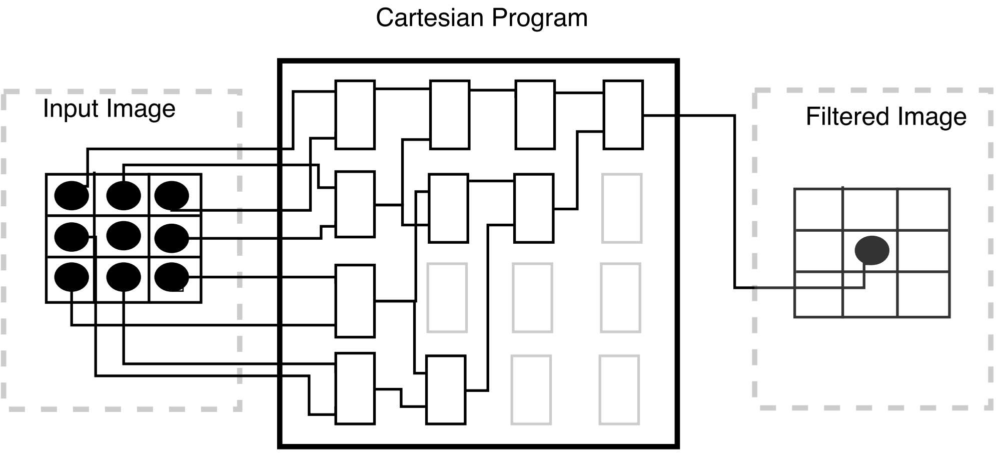


Figure 3.13: Illustration of an image operator evaluation procedure in CGP

design of specific noise reduction operators. The inputs of the genetic program are pixel values of a noisy image, and the output of the program is compared to the pixel values of a clear image. In this way, CGP can easily be used for the design-specific operators that fulfill a predefined standard. A visual example of the image operator design problem is given in Figure 3.13. The following presented technique for evolving and evaluating image operators is based on a procedure called *convolution*. Beside to the application of so called convolution filters and operators in image processing, the basic idea of convolution is commonly used in the field of *Convolutional Neural Networks* for image processing neural networks. In this reseach field, the method is known as *Padding*.

4 A Theoretical Analysis of CGP

In this chapter, a first theoretical runtime analysis of CGP will be introduced. The analysis is based on the use of two simple test problems called SUM and COUNTING OPERATORS. The runtime analysis of these problems is based on one of the *state-of-the-art* techniques for analyzing evolutionary algorithms. This technique is called *Multiplicative Drift Analysis*.

4.1 Drift Analysis

Drift analysis is one of the *state-of-the-art* techniques to analyze the runtime of randomized search heuristics, such as evolutionary algorithms. Furthermore, drift analysis is a powerful tool to analyze the optimization behavior of a randomized search algorithm over a search space by measuring the progress of the algorithm with respect to a *potential function*. Such a function maps each search point to a non-negative real number, where a potential of zero indicates that the search point is optimal. Drift analysis has significantly contributed to the analysis of meta-heuristics. Many significant results about the optimization time of meta-heuristics were achieved with drift analysis.

Multiplicative Drift Analysis

Multiplicative Drift Analysis as introduced by Doerr et. al. [18, 20, 19] is based on *Additive Drift Analysis* which has been proposed by He et al. [41, 40]. The multiplicative drift theorem can be considered as the multiplicative version of the additive drift theorem.

Theorem 1 (Additive Drift [41])**.** *Let $S \subseteq \mathbb{R}$ be a finite set of positive numbers and let $(X^{(t)})_{t\in\mathbb{N}}$ over S be a sequence of random variables over $S \cup \{0\}$. Let T be the random variable that denotes the first point in time $t \in \mathbb{N}$ for which $X^{(t)} = 0$.*
Suppose that there exists a constant $\delta > 0$ such that

$$E[X^{(t)} - X^{(t+1)} | T > t] \geq \delta \tag{4.1}$$

holds. Then

$$E[T] \leq \frac{X^{(0)}}{\delta} \tag{4.2}$$

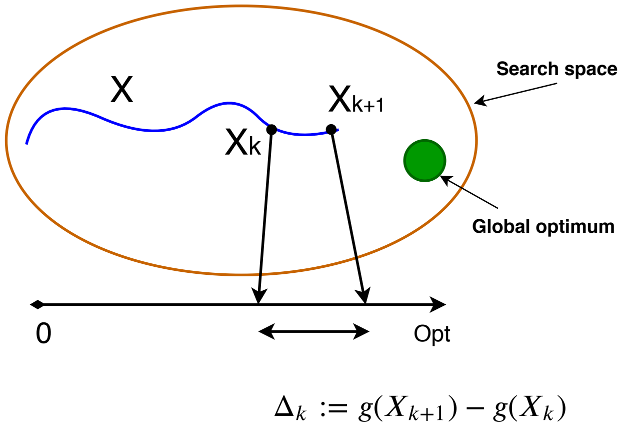


Figure 4.1: Illustration of drift measurement for a process X. X_k and X_{k+1} are two measurements of the *potential function* $g(x)$ over time. The value of Δ represents the drift of the process between the measurements.

The additive drift theorem describes how to relate the expected time at which the potential value reaches zero to the first time at which the expected value of the potential reaches zero. If the potential decreases in each step and in expectation by δ then after $X^{(0)}/\delta$ steps the expected potential is zero. To apply the previous theorem to the analysis of randomized search heuristics over a finite search space S, the defined potential function $h : S \to \mathbb{R}$ maps all optimal search points to zero and all non-optimal search points to values which are larger than zero. The random variable $X^{(t)}$ is defined as the potential $h(X^{(t)})$ of a search point $X^{(t)}$ in the t-th iteration of the algorithm. The random variable T is defined as the optimization time of the algorithm, which is the number of iterations until the algorithm finds an optimum.
When applying Theorem 1, the expected difference between $h(X^{(t)})$ and $h(X^{(t+1)})$ is called the drift of the random process $\{X^{(t)}\}_{t\in\mathbb{N}}$ with respect to h. This drift is *additive* if condition (4.1) holds.

The multiplicative method allows easier analyses in those settings where the optimization progress is roughly proportional to the current distance to the optimum. This method requires a progress which *multiplicatively* depends on the current potential value. That is the reason why the method was named multiplicative drift analysis. It has been found that for several problems, such potential functions are a natural choice [18, 20]. According to Doerr et al. [19]:

> ”Multiplicative drift analysis allows to largely separate the structural analysis of an optimization process from the actual calculation of a bound on the expected optimization time.”

[19, p. 2]

Furthermore, Doerr et al. [19] stated that:

> ”The runtime bounds obtained by multiplicative drift analysis are often sharper than those resulting from previously used techniques.”

[19, p. 3]

However, since multiplicative drift analysis is derived from the original additive result, it is clear that the multiplicative version cannot be stronger than the original theorem.

Theorem 2 (Multiplicative Drift [18])**.** *Let $S \subseteq \mathbb{R}$ be a finite set of positive numbers with minimum $s_{\min}$. Let $(X^{(t)})_{t\in\mathbb{N}}$ over S be a sequence of random variables over $S \cup \{0\}$. Let T be the random variable that denotes the first point in time $t \in \mathbb{N}$ for which $X^{(t)} = 0$. If there exists δ, $c_{\max}$, $c_{\min} > 0$ such that*

$$E[X^{(t)} - X^{(t+1)}|X^{(t)}] \geq \delta \cdot X^{(t)} \tag{4.3}$$

and

$$c_{\min} \leq X^{(t)} \leq c_{\max} \tag{4.4}$$

for all $t < T$, then

$$E[T] \leq \frac{2}{\delta} \cdot \ln(1 + \frac{c_{\max}}{c_{\min}}) \tag{4.5}$$

The drift of a random process concerning a potential function g is multiplicative if condition (4.3) holds for the affiliated random variables. The advantage of the multiplicative approach is that it allows using potential functions that are more natural. The most natural potential function can be considered as the distance of the objective value of the current solution to the optimum. This condition is a good choice in the analysis of combinatorial optimization problems [18, 20].

4.2 Single Active Gene Mutation Strategy

The single active gene mutation strategy as proposed by Goldman et al. [33] mutates at least one active gene of an individual in one generation. This means that all genes of active function nodes and the output nodes can be selected for mutation. The position of a gene of an active function node is chosen at random with a discrete uniform distributed random value:

Given a set G_a of active function nodes. Let $n_a = |G_\mathrm{a}|$ be the number of active function nodes of an CGP individual and let θ_a be an active function node. Given the number of function nodes n_f, the number of inputs n_i and the maximum arity a of the function nodes, θ_a can be formally described as a tuple $\theta_\mathrm{a} = \{x_a, g_f, g_\mathrm{c}^0, ..., g_\mathrm{c}^{a-1}\}$:

- $x_a \in \{x \in \mathbb{N} \mid n_\mathrm{i} \leq x \leq n_\mathrm{i} + n_\mathrm{f} - 1\}$ is the number of the function node
- $g_f \in \{x \in \mathbb{N}^0 \mid 0 \leq x \leq |F| - 1\}$ is the function gene
- $g_c^0, ..., g_c^{a-1} \in \{x \in \mathbb{N}^0 \mid 0 \leq x \leq x_\mathrm{f} - 1\}$ are the connection genes

Given a set G_o of output genes. Let $n_\mathrm{o} = |G_\mathrm{o}|$ be the number of output genes of an CGP individual. Each output gene can vary in the intervall $g_\mathrm{o} \in \{x \in \mathbb{N}^0 \mid 0 \leq x \leq n_\mathrm{i} + n_\mathrm{f} - 1\}$:
The number of active genes g_a can be calculated by $g_a = n_a * (1 + a) + n_\mathrm{o}$. The active gene which will be mutated is selected uniformly at random with probability $\frac{1}{g_a}$

The mutation on the selected gene of an active function node or output node is done by a flip of the gene value in the interval of a particular gene which is defined above. The starting index p_a (e.g. their index of the function gene) of an active function node in the genotype can be calculated by $p_\mathrm{a} = (x_\mathrm{a} - n_\mathrm{i}) * (a + 1)$ This procedure is equal to the standard probabilistic CGP mutation: The value of a connection gene is flipped between zero and a value less than the current node number, the value of a function gene is flipped with a value between zero to the maximum index of the function lookup table. Either a function gene and a connection gene of an active function node or an output gene can be mutated when the single active gene mutation strategy is performed.

4.3 Previous Theoretical Work on GP and CGP

The understanding of GP behavior on a theoretical level has been considered relatively poor in the past when compared to the number of successful practical applications [110, 126]:

> ”There remain a number of significant open issues despite the successful application of GP to a number of challenging real-world problem domains and progress in the development of a theory explaining the behavior and dynamics of GP.”

[110, p. 339]

However, the lack of theoretical results in the field GP was already known over 20 years ago:

> ”One of the most important deficiency in GP at the moment is the lack of sound theoretical foundations.”

[74, p. 28]

Furthermore, theoretical models for GP seem to be hard to obtain even if the complexity of GP algorithms can be considered as low [70]. A major theoretical contribution to the understanding of GP behavior has been made by applying schema theory [72, 127, 128, 125, 129]. However, the results of these works do not contribute to the runtime analysis of GP.

According to Mambrini et al. [88], the first studies of runtime analysis in GP focused on two functions, which are called ORDER and MAJORITY. For these types of problems, the fitness of an individual depends on the structure of the syntax tree and not on its execution. However, these types of problems can be considered as very simple compared to the problems to which GP is usually applied. However, according to Neumann et al. [107], the results of the mentioned problems show that GP can optimize both functions efficiently.

In their work, Mambrini et al. [88] reported that a recent study [65] analyzed the same simple GP systems on the MAX Problem. The analysis included a set of functions, a set of terminals, and a bound on the maximum depth of the solution. The goal is to evolve a tree that returns the maximum value given any combination of functions and terminals [65]. The results of the analysis show that simple GP systems can efficiently evolve MAX with a function set F=[+; $*$] and one constant as the terminal set. Compared to the previous functions, MAX is more similar to those evolved by GP in practical applications since the fitness indeed depends on the behavior of the computed function on the input. Still, dependence is not very strong, since the space of possible inputs can be partitioned into two subsets such that for every input in a subset, the optimal solution to the problem is the same.

Moraglio et al. [103] and Mambrini [104] obtained two more theoretical results by the runtime analysis of mutation-based Geometric Semantic Genetic Programming for evolving Boolean and basic regression functions.
Recently, Mambrini et al. [88] presented a theoretical analysis of two simple GP algorithms on two Boolean problems called AND and XOR. Both algorithms were equipped with a minimal function set with a maximum of two functions. It has been rigorously proved that both algorithms can solve both easy problems with minimal sets efficiently. However, Mambrini et al. [88] stated that:

> “If an extra function (i.e. NOT) is added to the function set, the algorithms require at least exponential time to evolve the conjunction of n variables. ”

[88, p. 99]

Recently, Lissovoi et al. [78, 77] presented results on the time and space complexity of GP for evolving Boolean conjunctions. The authors present a performance analysis that sheds light on the behavior of simple GP systems for evolving conjunctions of n variables (AND_n).

On the one hand the analysis of a random local search GP with minimal terminal and function sets revealed the relationship between the number of iterations and the expected error of the evolved program on the complete training set. The authors also considered a more realistic GP system equipped with a global mutation operator and proved that it could efficiently solve AND_n by producing programs of a linear size that fit a training set to optimality and with high probability generalize well. Based on the results of Lissovoi et al. , Doerr et al. [21] made a considerable step forward by analyzing the behavior and performance of the GP system for evolving a Boolean function with unknown components, i.e., the function may consist of both conjunctions and disjunctions. In their work Doerr et al. rigorously proved that if the target function is the conjunction of n variables, the random local search GP using the complete truth table to evaluate program quality evolves the exact target function in $\text{O}(\ell \log 2n)$ iterations in expectation, where $\ell \geq n$ is a limit on the depth of any accepted tree.

Regarding the theoretical knowledge of CGP, Woodward [165] investigated the functional complexity in CGP. To our best knowledge, the work of Woodward seems to be the only theoretical work contributed to the understanding of CGP behavior. Furthermore, Woodward's work does not contribute to the knowledge of the runtime complexity by obtaining upper and lower runtime bounds of the CGP algorithm itself. This significant lack of theoretical knowledge in CGP has been the motivation for our work.

4.4 Preliminaries

We will analyze a (1+1)-CGP algorithm on test problems called SUM and COUNTING OPERATORS. We say that an algorithm solves a problem efficiently if it can evolve a solution in expected polynomial time, where time is defined as the number of fitness function evaluations. As a genetic operator, the *single-active-gene* mutation strategy is in use. Since only one gene is chosen for mutation, the mutation can be performed on a connection gene or the function gene. The offspring replaces the parent only if the offspring has a better fitness value. We will analyze two scenarios. For the SUM problem, the runtime analysis of the algorithm depends on the number of n arity connections of a function node, which are represented by the connection genes of the CGP genotypes.

On the other hand, the runtime analysis of the algorithm depends on the number of n inputs (terminals) for the given COUNTING OPERATORS problem. We define *Artificial Fitness Levels* for the analysis of the SUM and COUNTING OPERA-

TORS problem. For our analysis, we use the *Multiplicative Drift Theorem*, which has been described in Section 4.2. For the analysis of the SUM problem, the CGP is equipped with a function set consisting of three mathematical functions, SUM, MIN, and AVG. For the analysis of the COUNTING OPERATORS problem, the function set only consists of the COUNT function.

For both problems, the number of function nodes is fixed and set to 1. The reason for this is that we will focus more on the theoretical analysis of the mutation abilities of CGP to build and reconnect arity connections. This behavior has been found as one of the key features of CGP and is considered highly beneficial for the efficiency of CGP. The output node is connected to the function node, which represents a configuration of the *levels back* parameter with a value of 1.

4.4.1 The SUM problem

The SUM problem is a very simple mathematical test problem for the theoretical analysis of CGP behavior. With the SUM problem, we will analyze the (1+1)-CGP algorithm depending on the number n of arity connections of a function node. For the analysis of this problem, the number of function nodes in the genotype is limited to 1, and the genotype has two input nodes. The first input node is a terminal with a constant value of $x = 0$, and the second input is a constant with a value of $y = 1$. The goal of this problem is to connect all arity connections of the function node to the second input and to sum up the "1" values. The genotype has one output which is connected to the function node. The function set F consists of three functions F={SUM, MIN, MAX}. In the first place, we have a function SUM, which adds up all values of the connected inputs of the function node. The function set also consists of a function MIN, which calculates the minimum of the given input values. The third function of the function set is a function AVG, which calculates the average of the input values. An example of the SUM problem is shown in Figure 4.2.

4.4.2 The COUNTING OPERATORS problem

The COUNTING OPERATORS problem is simple counting problem. With the COUNTING OPERATORS problem we will analyze the $(1 + 1)$-CGP algorithm depending on the length of n input nodes. The number of function nodes in the genotype is set to 1. The COUNTING OPERATORS problem represents a simple maximize count problem, which has the goal to build up connections between all input nodes and the function node. The goal of this problem is to connect all n input nodes to the given function node. The count of different inputs which are connected to the function node is used as the fitness value of an individual. The background of this problem is the design of boolean circuits for which CGP is commonly used. To design boolean circuits with CGP, it is necessary to evolve boolean functions, such as AND or OR. In this case, a function node, which performs a certain boolean

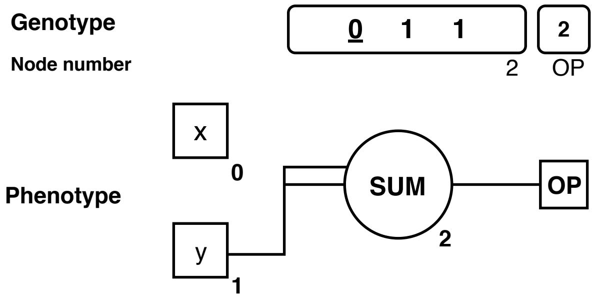


Figure 4.2: An example of the SUM problem which is used for the analysis. In the example, the function node has two arity connections and adds the input value of the second input node. The sum of this value is then given to the output.

operation has to be connected to previous input or function nodes. An example of the COUNTING OPERATORS problem is shown in Figure 4.3.

4.5 Analysis of the SUM problem

Theorem 3. *The (1+1)-CGP using n arity function node connections with a function set F={ SUM, MIN, AVG } of size $m := |F|$ solves SUM in expected time $\Theta(n \log n)$.*

Proof. First, we prove the upper bound using multiplicative drift analysis, cf. Proposition 1. Second, we prove the lower bound by estimating the probability that at least one connection to the first node does not switch to the second after a certain number of steps, cf. Proposition 2. □

Proposition 1. *The expected upper time bound for the SUM problem as defined above is $\mathcal{O}(n \log n)$.*

Proof. Let i be the number of arity connections which have not been connected to the second input node with the constant. The fitness of the individuals is defined by the value of the output. The fitness value depends on the respective function of the function node and the amount of arity connections which have been connected to the second input. A single connection gene is chosen with probability $1/(n+1)$. Therefore, the probability to achieve a higher fitness in a certain generation is $i/(n+1)$. The negative drift is 0 since solutions with fewer connections to the second input will not be accepted. We have

$$E[X^{(t)} - X^{(t+1)}|X^{(t)}] \geq \frac{i}{n+1} = \frac{1}{n+1} \cdot X^{(t)}.$$

Choosing $\delta = 1/(n+1)$, $c_{\min} = 1$, and $c_{\max} = n$ fulfills the requirements of Theorem 2. From it we obtain

$$E[T] \leq \frac{2}{\delta} \cdot \ln(1 + \frac{c_{\max}}{c_{\min}}) = 2 \cdot (n+1) \cdot \ln(1 + \frac{n}{1}) = \mathcal{O}(n \log n).$$

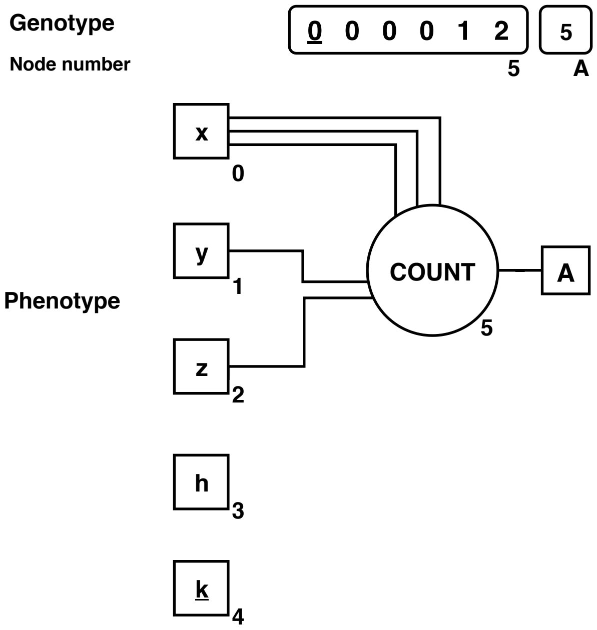


Figure 4.3: An example of the COUNTING OPERATORS problem which is used for the analysis. In the example, the function node has five arity connections and has build up connections to all five inputs nodes. The number of different inputs are counted by the function node which performs a COUNT function. The COUNT function determines the number of different input nodes which are connected to the function node.

For now, we did not consider the function gene. We analyze the expected time independently from the connection genes. With probability $1/(n+1)$ the function node is chosen for mutation. If SUM is not the current function operator and at least one arity connection is connected to the second input, then with probability $\frac{1}{2}$ the function operator is mutated to SUM. The probability that at least one arity connection is connected to the second input is at least $1 - \frac{1}{2^n}$. If SUM is the current operator, the function SUM is kept as executing function. The reason for this is that neither the use of the MIN function nor the use of the AVG function can achieve a higher fitness value than the SUM function. Assuming the SUM operator has not yet been chosen, then the probability to mutate to SUM is at least $\frac{1}{n+1} \cdot \frac{1}{2} \cdot (1 - \frac{1}{2^n})$, which implies an upper bound for the expected number of turns to mutate to the SUM operator of $\mathcal{O}(n)$. □

Proposition 2. *The expected lower time bound for the SUM problem, as defined above, is* $\Omega(n \log n)$*.*

Proof. We assume the function node is set to SUM during initialization; the expected running time with random initialization of the function node cannot be lower. A given connection flips with probability $p := 1/(n+1)$. It does not flip in t steps with probability $(1-p)^t$. Therefore, each of $n/2$ inputs switch at least once in t steps with probability $(1-(1-p)^t)^{n/2}$. With p as above and $t := n \log n$ we have

$$\left(1-(1-p)^t\right)^{\frac{n}{2}} = \left(1-\left(1-\frac{1}{n+1}\right)^{n\log n}\right)^{\frac{n}{2}}$$

$$\leq \left(1-\left(\frac{1}{e}\right)^{\log n}\right)^{\frac{n}{2}} = \left(1-\frac{1}{n}\right)^{n\cdot\frac{1}{2}} \leq \left(\frac{1}{e}\right)^{\frac{1}{2}} < 0.61$$

Therefore, with constant probability $c > 1 - 0.61 = 0.39$ at least one of $n/2$ inputs does not switch after t steps.

With probability, at least $1/2$ at least $n/2$ connections are initialized to the first input. This follows from the binomial distribution. With the results above we obtain the following estimation on the lower bound.

$$E(T) = \sum_{t=1}^{\infty} t \cdot p(t) \geq n \log n \cdot \frac{1}{2} \cdot 0.39 = \Omega(n \log n)$$

□

4.6 Analysis of the COUNTING OPERATORS problem

Theorem 4. *The (1+1)-CGP using n input nodes and* $F = \{$ *COUNT* $\}$ *solves the COUNTING OPERATORS problems in expected time* $\mathcal{O}(n^2 \log n)$*.*

Proof. Let i be the number of input nodes that have not been connected to the function node of the genotype. The output of an individual is the number of inputs which are connected to the function node. This number represents the fitness of the individuals. To classify the fitness of an individual more precisely, we define *Artificial Fitness Levels* $(A_1, A_2, A_j \dots, A_n)$, where j is the number of input nodes which have been connected to the function node. We observe that in level A_j, at least $i+1$ connections share another connection to the same input node. A higher artificial fitness level is achieved, if and only if such a connection is mutated to an unconnected input. The probability to mutate to an unconnected input is $i/(n-1)$. In level A_n, all n input connections have been connected to the function node, and the COUNTING OPERATORS problem is solved. We again use multiplicative drift analysis to prove the upper bound. The negative drift is 0 since solutions with more unconnected nodes will not be accepted. We have

$$E[X^{(t)} - X^{(t+1)} | X^{(t)}] \geq \frac{i+1}{n} \cdot \frac{i}{n-1}$$

$$= \frac{i+1}{n(n-1)} \cdot i \geq \frac{2}{n^2} \cdot i = \frac{2}{n^2} \cdot X^{(t)}.$$

Choosing $\delta = 2/n^2$, $c_{\min} = 1$, and $c_{\max} = n-1$ fulfills the requirements of Theorem 2. From it we obtain

$$E[T] \leq \frac{2}{\delta} \cdot \ln(1 + \frac{c_{\max}}{c_{\min}}) = n^2 \cdot \ln(1 + \frac{n-1}{1}) = \mathcal{O}(n^2 \log n).$$

□

Theorem 5. *The (1+1)-CGP using n input nodes and $F = \{$ COUNT $\}$ solves the COUNTING OPERATORS problem in expected time $\Omega(n^2)$.*

Proof. The lower bound of $\Omega(n^2)$ is obvious. The probability to start in fitness level A_{n-1} is at most $1/2$, if $n > 1$. The probability to proceed from fitness level A_{n-1} to A_n is $2/n \cdot 1/n$, therefore the expected time is $\Omega(n^2)$.

□

4.7 Discussion

The results of our time complexity analysis show that CGP can solve the SUM problem in expected time $\Theta(n \log n)$. For the COUNTING OPERATORS problem, we proved an upper bound $\mathcal{O}(n^2 \log n)$ and a lower bound $\Omega(n^2)$. If a function is part of the function set, which cannot lead to the correct solution, CGP can efficiently solve the SUM problem. Compared to the conventional tree representation of GP, the graph-based representation enables multiple connections between former nodes and the inputs. Consequently, the probabilities that beneficial mutations are performed, and the algorithm proceeds toward the global optimum can be quite low. Therefore, the result for the SUM problem, when the function set includes functions that do not contribute to the evolutionary search, is quite interesting. Regarding the analysis of Mambrini et al. [88] that found that if an extra function was added to the function set, the algorithms require at least exponential time to evolve the simple Boolean problems, our result sheds more light on the behavior of CGP when such function sets are in use on the SUM problem. However, we want to point out that our finding cannot be generalized yet as we only analyzed one problem on this behavior. According to the findings of Mambrini et al. [88], it may be possible that such problems also exist in CGP.

One point which should be discussed is the use of the single active gene mutation strategy. This strategy has been found as more beneficial for the search performance of CGP as the use of classical mutation probabilities on a practical level. However, flipping merely one gene may reduce the probability that a mutation is performed, which hopefully processes the algorithm toward the global optimum. Moreover, the use of the single active gene mutation strategy has only been investigated and compared on an experimental level. Therefore, we think a theoretical analysis of a $(1 + 1)$-CGP algorithm with classical mutation probabilities is needed and should be considered in future work.

Another point which should be discussed is the fact that both test problems only include one function node. As a first step foward, we focused on the behavior and efficiency of the point mutation operator. Especially in terms of building and reconnecting connections between input nodes and arity connection genes. This behavior has been considered highly important for the search performance of CGP but has never been investigated on a theoretical level. For the COUNTING OPERATORS problem, we proved a higher upper bound as for the SUM problem. The results indicate that the expected time of CGP can significantly increase when the given problem enables a high number of combinatorial possibilities.

The last point which should be discussed is the complexity of the test problems itself. From a practical point of view, these problems can be considered as toy problems that have the limitation of being very simple and with characteristics of regularity that makes them rather different from any real-life application or practical problem.

Furthermore, compared to the state of theoretical knowledge in tree-based GP, our analysis with the introduced test problems is quite simple. For instance, Mambrini et al. also investigated incomplete training sets. Nevertheless, as a first step forward, we focused more on the development of suitable test problems and studied the feasibility of runtime complexity analysis in CGP. The complexity of our problem can easily be increased for further analyses. For instance, the COUNTING OPERATORS problem can be extended to a COUNTING NEGATED OPERATORS problem. To solve this problem, two functions (COUNT & NOT) are necessary, and at least two function nodes are needed to find the correct solution. Therefore, the analysis of the COUNTING NEGATED OPERATORS problem would be a natural next step.

4.8 Conclusion

A first time complexity analysis for CGP has been presented. We introduced a simple mathematical test problem and a simple Boolean test problem for CGP, which can be used for the drift analysis of the $(1+1)$-CGP algorithm. Our analysis has shown that CGP can solve the mathematical SUM problem in time $\Theta(n \log n)$. Furthermore, adding functions to the function set, which do not contribute to the evolution of the correct solution does not degrade the time complexity of the $(1+1)$-CGP for this problem. However, for the COUNTING OPERATORS problem, we proved an upper bound of $\mathcal{O}(n^2 \log n)$ and a lower bound of $\Omega(n^2)$. Our result clearly shows that even a simple Boolean problem can lead to a significant level of complexity in CGP, which makes it difficult to find the ideal solution in polynomial time.

5 On The 1+4 Dogma of CGP

5.1 Introduction

The first systematic investigation on an efficient optimization scheme for CGP was done by Miller 1999 in [97]. Miller studied the behavior of a canonical EA with a genotypic uniform recombination operator and a $(1 + \lambda)$ selection scheme. He configured CGP as a square grid of functional nodes with the maximal length of feed-forward wires of two. In 1999, it was already known that a “neutral selection” scheme that is preferring offspring individuals for propagating into the next generation if they are on par or better than the parent individual is highly beneficial for CGP. In a series of experiments, Miller observed that the evolution of digital circuits using CGP can be solved better by local search-like approaches employing “neutral selection” than by the canonical EA deployed by Miller. Miller also concluded that recombination only marginally contributes to the search performance of CGP.

In this chapter, we address the question, whether the popular choice of $(1 + 4)$ selection scheme in combination with the single-line CGP genotype can be generalized. For this, we rely on an unbiased parameter tuning method to identify (i) well-performing parameterizations of CGP and (ii) efficient optimization schemes.

While initially CGP’s selection scheme has often been labeled as a (1+4) Evolutionary Strategy (ES) in the CGP literature, we would like to use a different and a more precise notion in this chapter. ES is a well-known method for real-valued numerical optimization incorporating auto-adaptation mechanisms. CGP borrows from the conventional ES only the selection scheme, i.e., using μ parent individuals to produce λ offspring individuals and select the best individuals for the next generation deterministically. While the original (1+4) selection scheme could be seen as a very close derivative of the regular Hill Climbing (HC), i.e., one could use the “(1+4) HC” notion, in this chapter, we are also investigating general $(\mu + \lambda)$ selection schemes. These schemes are not simple single-trajectory style algorithms, as realized by the conventional HC, and we, therefore, switch to the more general **“$(\mu + \lambda)$ CGP”** notion to label the search strategy.

Discrete and combinatorial search spaces usually lack understanding for their topological properties and the notion of a “gradient”, which would allow using efficient techniques common to discrete and continuous search spaces. Furthermore, information drift among individuals cannot be implemented easily, rendering global methods relying on a recombination operator infeasible. In consequence, trajectory-based local-search techniques are typically used to solve combinatorial problems.

An incomplete illustration exemplary of some famous trajectory-based metaheuristic families regarding their algorithmic principles for escaping local optima is presented

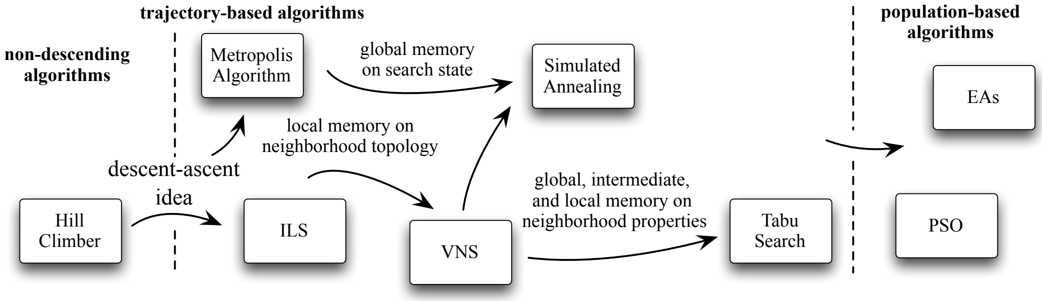

trajectory-based algorithms
non-descending algorithms
population-based algorithms
Metropolis Algorithm
global memory on search state
Simulated Annealing
EAs
local memory on neighborhood topology
descent-ascent idea
global, intermediate, and local memory on neighborhood properties
Hill Climber
ILS
VNS
Tabu Search
PSO

Figure 5.1: An incomplete exemplary illustration of some popular trajectory-based metaheuristic families regarding their algorithmic principles for escaping local optima.

in Figure 5.1. On the left-hand side of Figure 5.1, HC is presented as the most straightforward method that implements no or only rudimentary measures, such as neutral drift. HC accepts no solutions during the search that are worse than the best solution found so far. When relaxing this strict rule, Iterated Local Search (ILS) and variants of the Variable Neighbourhood Search implement search restart from a solution selected randomly in the proximity of a solution found so far. The Metropolis Algorithms (MA) and Simulated Annealing (SA) are extending this idea by regularly accepting worse solutions at some probability in the hope of reaching remote search spaces goes one step further and deterministically receives the best solution in the neighborhood of current solution. However, the refinement of these techniques for escaping local optima not necessarily results in faster convergence rates. In previous work, we could observe HC excelling for small optimization functions, while for larger goal functions, more elaborate techniques, such as SA, are converging better. A reason for this can be that the selection schemes of SA, Tabu Search (TS)... allows the algorithms in the final search stages to escape from equipotential fitness plateaus more effectively.

5.2 Computational Effort

The computational effort (CE) is a measure introduced by Koza in [67]. The CE statistic is used to report the amount of computational effort to solve a problem with a 99% probability with a GP System. We use the minimum of the computational effort, as shown in Equation (5.5). This methodology has been used by Koza to describe experiments on several problems.

To calculate this measure, we first have to define a cumulative probability of success $P(N,l)$ which represents the number of runs ($N_s(l)$) that have been successful after l generations in relation to the number of totals runs R_{total} by using N individuals in each run.

$$P(N,l) := \frac{N_s(l)}{R_{total}}. \tag{5.1}$$

Since we hope that our GP System can solve the problem with 99% probability, we have to determine the number of runs that are required to find the solution. However, because of premature convergence, a 99% probability of success in every run may never occur. If R runs are independent, the odds of failure for all runs can be calculated by

$$P_{\text{all fail}} := (1 - P(N, l))^R \tag{5.2}$$

and with odds of failure, we can compute the number of independent runs $R(z)$, which are required to get a solution with a confidence interval of z where z is often chosen as a probability of 99%.

$$R(z) := \left\lceil \frac{\log(1 - z)}{\log(1 - P(N, l))} \right\rceil \tag{5.3}$$

The CE statistic $I(N, z, l)$ describes the number of evaluations that have to be performed to solve a problem to a proportion of z. This is done by multiplying the total number of individuals processed at the end of generation number l to $R(z)$.

$$I(N, z, l) := (N \cdot l \cdot R(z)) \tag{5.4}$$

Koza defined the statistic overall generation numbers l to find the minimum computational effort $I_{\text{min}}(N, z)$ to solve a given problem. For the determination of $I_{\text{min}}(N, z)$ we take the minimum of all sampled individuals $I(N, z, l)$ as shown in Equation (5.5).

$$I_{\text{min}}(N, z) := \arg \min_{l} (N \cdot l \cdot R(z)) \tag{5.5}$$

5.3 Common Parametrizations of Cartesian Genetic Programming

The origins of knowledge about the efficient use of CGP can be found in an empirical study by Miller [97]. Four Boolean functions were used to investigate the effectiveness of CGP with different parameters which are highly relevant for the use of CGP. The empirical study focused on the investigation of different settings for the geometries of the CGP representation and the population size. A genetic algorithm was used for CGP with a uniform crossover operator and point mutation. Miller investigated geometries of 4 x 4 up to 30x30. For investigating the population size, a range with a minimum of 4 and a maximum of 50 individuals was used for each geometry setting. Related to the population size, it was found that small populations perform most effectively on the tested functions, which was one of the primary research questions of the study. Moreover, a population size of 3 up to 6 individuals performed best on the tested problems whereby the setting the of 4 individuals performed best most often in this range. However, Miller showed that when the maximum number of allowed nodes was tiny, the dependence of computational effort with population size is reversed. Another key finding of the study was the fact that recombination does only seem to contribute marginally to the search performance of the evolutionary search.

Yu and Miller [166] continued the use of very low population sizes with a (1+4) CGP by the study of neutrality in the context of CGP when applied to Boolean function learning. The study on Even-Parity benchmarks outlined a positive relationship between neutrality and evolvability, and the authors concluded that neutrality improves evolvability.

In later work, Yu and Miller [167] applied the concept of neutrality to four needle-in-a-haystack even-parity benchmarks. The combination of neutrality and the chosen (1+λ) CGP has been found as very successful in solving these complex functions. Furthermore, the configuration also included a high level of genotypic redundancy of about 100 function nodes, which has been found successful for solving the four needle-in-a-haystack functions.

CGP has been successfully extended for the Automatic Definition and reuse of Functions (ADF) by Walker et al. [156] and Kaufmann et al. [61], which is known as *Embedded Cartesian Genetic Programming* (ECGP). For the ECGP, Walker used (1+4) CGP. Kaufmann et al. adopted module creation for cone- and age-based recombination schemes elaborated on a Genetic Algorithm (GA) with medium population sizes and compared it to (1+4) CGP. The GA with cone-based crossover showed good performances for functions with repetitive inner patterns, such as arithmetic operations, while the age-based recombination excelled for functions with more randomized internal structures, such as pattern matching kernels.

Miller et al. investigated the role of genotypic length and mutation rates in CGP with more detailed experiments in [99]. The best performances were identified for large genotypes and low mutation rates. However, the study only included a small set of Boolean functions.

Later work focused on a more detailed investigation of the neutral genetic drift with the use of CGP. The authors used (1+4) CGP, which we assume as the best setting for the experiments.

Our survey on the most significant work , which contributed to the understanding and development of CGP shows that $(1 + 4)$-CGP seems to be a popular choice. Moreover on a set of boolean function problems the $(1 + 4)$-CGP performed best when compared to other settings of the λ . However, the results from previous work open the question if the use of the $(1+4)$-CGP with a extremely large genotype can be generalized. Furthermore, it is not answered yet, if small population sizes are the right choice for CGP in general since former experiments only focused on boolean function problems. This motivates our research.

Benchmarks	Functional set
$(i, i, 1)$-add, (i, i)-mul	$a \wedge b$, $a \wedge \bar{b}$, $\bar{a} \wedge b$, $a \oplus b$, $a \vert b$
even parity	$a \wedge b$, $a \wedge \bar{b}$, $\bar{a} \wedge b$, $a \vert b$, $a \vert \bar{b}$, $\bar{a} \vert b$
Koza	$+$, $-$, $*$, $/$, sin, cos, $\ln(\vert n \vert)$, e^n
Keijzer	$+$, $*$, n^{-1}, $-n$,$\sqrt{n}$

Table 5.1: Functional set of all benchmarks.

5.4 Experimental Setup

The basic methodology of this work is that we first define a set of goal functions $f_1, f_2 \ldots$ and a set of optimization algorithms $a_1, a_2 \ldots$. Then, we execute for each tuple (f_i, a_j) an automatic parameter-tuning tool. The resulting configurations are evaluated at the end in separate experiments to derive their convergence behaviors and for comparison. In this section, we are describing the selected benchmarks, optimization algorithms, and the parameter-tuning setup.

5.4.1 Boolean Benchmarks

We evaluate CGP parameterizations for pure combinatorial and partially continuous sets of goal functions.
Boolean circuits:
The first class of benchmark functions consists of the Boolean adder, multiplier, and even parity functions. An $(n, n, 1)$ adder computes the $n + 1$-bit sum of two n-bit binary encoded numbers and an carry-in wire. An (n, n) multiplier computes the $2n$-bit binary-encoded product of two n-bit binary-encoded numbers. An n-bit even parity circuit computes an output bit such that when counting all 1-bits in the inputs and outputs, their sum is even. The set of 2-input Boolean functions that may be used as functional nodes in CGP genotypes is presented in the first rows of Table 5.1. An experiment is stopped if a perfect solution has been found, or the maximal number of fitness evaluations has been exceeded.

5.4.2 Regression Benchmarks

The second set of benchmark consists of twelve symbolic regression functions (Koza-2, -3, Nguyen-4 . . . -10, Keijzer-4, -6, Pagie-1) from the work of McDermott et al. [90]. The functions are shown in Table 5.2. A training data set U$[a, b, c]$ in Table 5.2 consists of c uniformly sampled from an interval $[a, b]$. E$[a, b, c]$ is defined as a set of c equidistantly sampled numbers in the same interval. For the Keijzer-6 problem the upper bound of the summation x was calculated as the floor of the current data point. The cost function is defined as the sum of absolute differences between functional values of the reference and evolved function at the data points of the training set. An experiment is terminated if the cost function reaches a value below or equals to 0.01, or the maximal number of fitness evaluations has been exceeded. We omit to investigate the Koza-1 benchmark ("quartic") and resort to Keijzer-6, Nguyen-7 and Pagie-1, which have been proposed as a replacement for Koza-1 by White et al. [160]. The functional sets for Koza and Keijzer functions are presented in the last two rows of Table 5.1.

Table 5.2: Symbolic regression benchmarks

Goal function	Objective Function	Vars	Training Set
Koza-2	$x^5 - 2x^3 + x$	1	U[-1,1,20]
Koza-3	$x^6 - 2x^4 + x^2$	1	U[-1,1,20]
Nguyen-4	$x^6 + x^5 + x^4 + x^3 + x^2 + x$	1	U[-1,1,20]
Nguyen-5	$\sin(x^2)\cos(x) - 1$	1	U[-1,1,20]
Nguyen-6	$\sin(x) + \sin(x + x^2)$	1	U[-1,1,20]
Nguyen-7	$\ln(x + 1) + \ln(x^2 + 1)$	1	U[0,2,20]
Nguyen-8	$\sqrt{x}$	1	U[0,4,20]
Nguyen-9	$\sin(x^2) + \sin(y^2)$	2	U[-1,1,20]
Nguyen-10	$2 * \sin(x) * \cos(x)$	2	U[0,2,20]
Keijzer-4	$x^3 * e^{-1} * \cos(x) * \sin(x)(\sin^2(x) * \cos(x) - 1)$	1	E[0,10,0.05]
Keijzer-6	$\sum_{i=1}^{x} \frac{1}{i}$	1	E[1,50,1]
Pagie-1	$1/(1 + x^{-4}) + 1/(1 + y^{-4})$	2	E[-5,5,0.4]

5.4.3 Optimization Algorithms

As the baseline method, we select the (1+4) CGP, which is used in related work predominantly. The second and third algorithms are $(1 + \lambda)$ CGP and $(\mu + \lambda)$ CGP, where the number of offspring individuals λ and the number of parents μ are subject to optimization. For all CGP schemes, "neutral selection" has been realized. For optimizing Boolean circuits, we have additionally selected Simulated Annealing (SA) [117, 63, 49, 153] with the following cooling strategy:

$$A \leftarrow \frac{(T_{\text{start}} - T_{\text{end}})(N + 1)}{N}; \qquad B \leftarrow T_{\text{start}} - A; \qquad T_t \leftarrow \frac{A}{t + 1} + B.$$

T_{start} is the start temperature, T_{stop} is the stop temparature. N is the maximum number of iterations and t is the iteration index. Consequently, T_t is the temperature in the t-th iteration. In our experiments, we have evaluated eight temperature control strategies, presented in Table 5.3, and selected the third cooling scheme $T_t^{(3)}$ for further experiments, as it showed the best results for Boolean functions. Random sampling and random walk have also been investigated in preliminary experiments and sorted out because of inferior results. A visualization of the temperature control strategy $T_t^{(3)}$ is shown in Figure 5.2.

Table 5.3: SA temperature control strategies. T_0, T_N, and T_t are the start, terminal, and current temperatures. N is the maximal number of SA iterations.

$T_t^{(1)}$	$\leftarrow T_0 - \frac{T_0 - T_N}{N}$
$T_t^{(2)}$	$\leftarrow T_0 \left(\frac{T_n}{T_0}\right)^{\frac{t}{N}}$
A B $T_t^{(3)}$	$\leftarrow \frac{(T_{\text{start}} - T_{\text{end}})(N+1)}{N}$ $\leftarrow T_{\text{start}} - A$ $\leftarrow \frac{A}{t+1} + B$
$T_t^{(4)}$	$\leftarrow 0.5(T_0 - T_N)(1 + \cos(\frac{\pi t}{N})) + T_N$
$T_t^{(5)}$	$\leftarrow 0.5(T_0 - T_N)(1 - \tanh(\frac{10t}{N} - 5)) + T_N$
$T_t^{(6)}$	$\leftarrow \frac{T_0 - T_N}{\cosh(\frac{10t}{N})} + T_N$
$T_t^{(7)}$	$\leftarrow T_0 \exp(-\frac{1}{N} \ln(\frac{T_0}{T_N}) t)$
$T_t^{(8)}$	$\leftarrow T_0 \exp(-\frac{1}{N^2} \ln(\frac{T_0}{T_N}) t^2)$

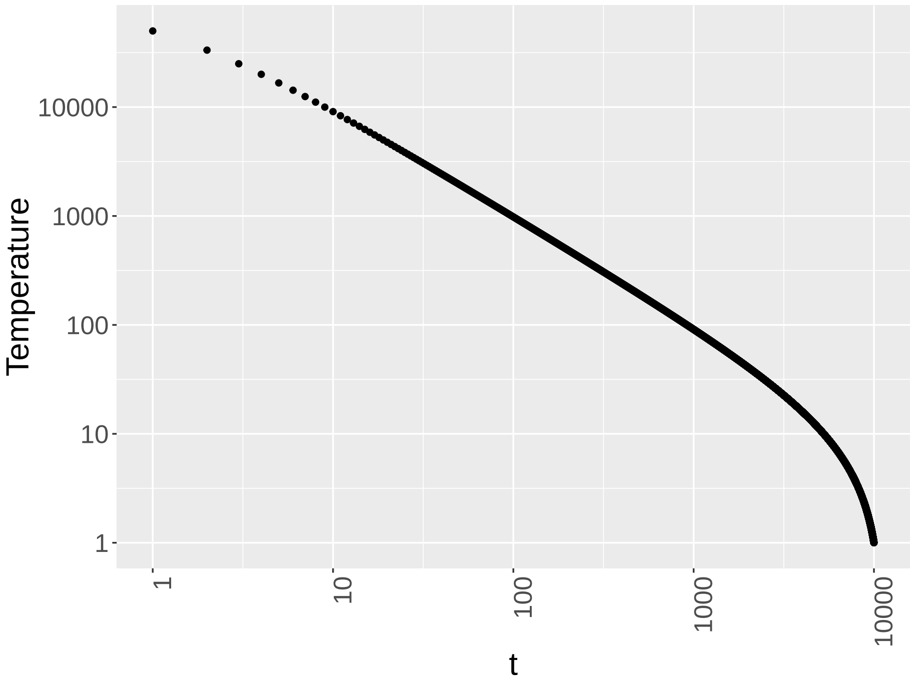


Figure 5.2: Visualization of temperature control strategy $T_t^{(3)}$ with $T_a = 100000$, $T_e = 1$ and $N = 10000$.

5.4.4 Automatic Parameter Tuning

To detect suitable parameterizations, we are using the iRace package [83]. We ported it to the `torq` batch system and parallelized the computations on the PC^2 cluster at the Paderborn University. For Boolean benchmarks iRace v2.1 and regression benchmarks iRace v0.7 were used. iRace was configured to execute 2000 trials for each tested algorithm-benchmark pair. Each trial consisted of multiple candidate algorithm runs. Boolean benchmarks returned the median number of fitness evaluations required to evolve a correct solution over three runs or the maximal positive integer in case the evolution failed. Regression benchmarks reported the median fitness over seven runs.

The parameter space explored by iRace is presented in Table 5.4. We extended the definition range of the number of columns of the $(1+4)$-CGP to 2000 for the $(3,3)$-multiply and $(8,9)$-parity benchmark after observing that iRace always evolving configurations with a maximal number of columns allowed so far (300). For the remaining experiments, the maximal number of columns remained unchanged. iRace usually evolves multiple good-performing configurations for an algorithm-benchmark pair. To verify the results of iRace, we have computed for each configuration, the median performance in 100 runs. For small benchmarks, as the (2,2,1)-add and (2,2)-mul, 10000 runs have been executed. We have then selected for each algorithm-benchmark pair the best performing configuration and report it here. For the tuning of the SA parameters, we defined the rule that the starting temperature T_{start} had to be smaller than the stopping temperator T_{stop}. Otherwise, the setting was rejected.

5.5 Results

The results of our experiments are presented in Table 5.5 and Table 5.6. Before going into details, we would like to shortly describe why we are reporting median fitness values instead of averages. For goal functions with a “correctness” property, i.e. Boolean circuits or regression functions with a functional quality requirement, an optimization algorithm may reach the maximal number of fitness evaluations without finding a valid solution. Computing the average number of function evaluations for finding a solution with the desired properties is, however, valid only if, in all runs, the goal fitness has been reached. Additionally, one must test first, whether the numbers are normally distributed. To avoid these issues, we report the median number of fitness evaluations for evolving a solution with a desired functional quality. For better interpretability, we also use the CE metric at $z = 99\%$. It is widely criticized but more useful for us than significance tests and effect measure sizes, because CE allows for a more direct comparison of differences in algorithm performances.

Table 5.4: Parameter space explored by iRace.

Boolean benchmarks, CGP

rows	1, 2, 3, 4, 6, 8, 10, 14, 20, 30, 50, 100
mut.[%]	(0.1, 7.0)
l	∞
μ, λ	1, 2, 3, 4, 8, 16, 32
columns	4, 6, 8, 10, 15, 20, 30, 50, 70, 100, 150, 200, 300

Boolean benchmarks, (1+4) CGP, $(3,3)$-mul, $8,9$-parity

columns	4, 6, 8, 10, 15, 20, 30, 50, 70, 100, 150, 200, 300, 400, 500, 600, 700, 800, 1000, 1500, 2000

Boolean benchmarks, SA

T_{start}	(0.001, 10000)
T_{stop}	(0.0000001, 1.0)

Regression benchmarks, CGP

rows	1, 2, 3, 4, 6, 8, 10, 14, 20, 30, 50, 100
columns	4, 6, 8, 10, 15, 20, 30, 50, 70, 100, 150, 200, 300
μ	1, 2, 3, 4, 8, 16, 18, 22, 32
λ	2, 4, 6, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192
mut.[%]	1, 2, 5, 7, 10, 15, 20

Regression benchmarks, SA

T_{start}	(0.001, 10000)
T_{stop}	(0.0000001, 1.0)

5.5.1 Evolution of Boolean Circuits

In Table 5.5, results for the Boolean benchmarks are presented. The table is subdivided vertically by benchmarks. In the “evolved parameters” columns, the CGP and algorithmic parameters found by iRace are presented. Dashed cells indicate that the according parameters are either irrelevant or fixed and have not been optimized by iRace.

The first observation that can be made is that the baseline (1+4) CGP on a single-line CGP is never a clear winner regarding the median number of fitness evaluations when evolving functionally correct Boolean circuits (c.f. Table 5.5). Except for the smallest benchmarks, the $(2,2,1)$-adder and the $(2,2)$-multiplier, and for the 8-parity benchmark, SA is always a clear winner. For the 8-parity benchmark SA is passed by $(\mu+\lambda)$ CGP only by roughly 2%. For more extensive benchmarks, like the

Table 5.5: Evaluation of CGP parameters for Boolean functions. Not optimized parameters are marked with an "–". The comparison prefers conventional (1+4) CGP, as iRace budget is set to 2000 for all configurations and challengers have more parameters to optimize. The results are measured in number of fitness evaluations. Best results are printed in *bold.* n_c and n_r - number of CGP columns and rows; m - mutation rate; T_{start} and T_{stop} - starting and stopping temperatures for SA with cooling shedule $T_t^{(3)}$. CE at $z = 99\%$.

goal function	algo-rithm	evolved parameters							termination[no. evaluations]			Comp. Effort	restart at eval.
		n_c	n_r	μ	λ	m[%]	T_{start}	T_{stop}	1Q	median	3Q		
(2,2,1) add	1+4 CGP	200	–	–	–	2.1112	–	–	14916	26532	49840	160753	91840
	1+λ CGP	100	200	–	3	0.3215	–	–	11316	18933	28797	89280	34350
	μ+λ CGP	200	50	1	1	0.3803	–	–	8114	**13129**	21723	**67860**	19849
	SA	200	2	–	–	1.8976	1299	0.0348	12242	20052	35411	109530	42284
(3,3,1) add	1+4 CGP	200	–	–	–	2.1512	–	–	113168	194120	326156	**689115**	689112
	1+λ CGP	150	1	–	3	1.9464	–	–	105789	178344	302211	929794	581961
	μ+λ CGP	100	4	1	3	0.8396	–	–	122460	190539	330936	1018919	451407
	SA	70	4	–	–	1.3706	4671	0.4366	88335	**149817**	246126	750368	621896
(4,4,1) add	1+4 CGP	200	–	–	–	1.2341	–	–	424924	697152	1182452	2830424	2404400
	1+λ CGP	300	2	–	2	0.6852	–	–	303080	501550	698950	2206982	1680482
	μ+λ CGP	100	4	1	1	1.1503	–	–	364545	545438	936699	2469195	2097544
	SA	150	3	–	–	0.6693	3610	0.6437	283038	**400832**	723341	**2034761**	1422236
(2,2) mul	1+4 CGP	100	–	–	–	2.9542	–	–	3452	5564	9136	28434	14864
	1+λ CGP	100	100	–	3	0.8680	–	–	2121	3417	5474	**16512**	9009
	μ+λ CGP	100	30	1	1	1.4332	–	–	2079	**3322**	5465	17349	7279
	SA	30	14	–	–	2.4941	58	0.0889	2661	4183	6801	21275	9959
(3,3) mul	1+4 CGP	2000	–	–	–	0.5008	–	–	274228	447220	722280	2103815	1787156
	1+λ CGP	200	20	–	2	0.2988	–	–	149824	288368	459822	1203021	1203020
	μ+λ CGP	150	30	1	2	0.2971	–	–	130250	224178	498888	1382722	361496
	SA	200	100	–	–	0.1622	3336	0.0870	84844	**148145**	356305	**949607**	169289
7-parity	1+4 CGP	300	–	–	–	1.2582	–	–	175628	271048	427788	1347746	645976
	1+λ CGP	300	8	–	2	0.7142	–	–	100408	186250	262668	762572	381284
	μ+λ CGP	300	2	1	2	0.9089	–	–	118996	186674	291118	696589	696588
	SA	150	8	–	–	0.7584	1528	0.2000	87773	**140463**	238599	**539214**	458054
8-parity	1+4 CGP	2000	–	–	–	0.9057	–	–	336420	461948	739504	2113156	1374636
	1+λ CGP	200	6	–	3	1.0381	–	–	310524	486894	798396	2408346	932859
	μ+λ CGP	300	6	1	1	0.5578	–	–	192417	**323192**	455204	1404562	702280
	SA	300	4	–	–	0.6733	417	0.3479	213877	329472	479532	**1196482**	1196482
9-parity	1+4 CGP	2000	–	–	–	0.8718	–	–	628536	1011220	1718660	5380705	1487336
	1+λ CGP	150	3	–	2	0.7050	–	–	617418	959194	1570728	2859287	2859286
	μ+λ CGP	300	3	1	1	0.8519	–	–	512420	755543	1239866	3073095	1774561
	SA	300	10	–	–	0.3784	2209	0.2907	392406	**579111**	910828	**2209561**	1876989

Table 5.6: Evaluation of CGP parameters for symbolic regression functions. Not optimized parameters are marked with an "–". The comparison prefers conventional 1+4 CGP, as iRace budget is set to 2000 for all configurations and challengers have more parameters to optimize.

goal function	optimization algorithm	optimized parameters					best fitness quartiles			Success Rate
		n_c	n_r	μ	λ	m[%]	1Q	2Q	3Q	
Koza-2	$1 + 4$ CGP	150	–	–	–	5	0.0095	0.0099	0.0325	0.65
	$1 + \lambda$ CGP	150	3	–	128	2	0.0091	0.0098	0.0364	0.68
	$\mu + \lambda$ **CGP**	**150**	**3**	**18**	**2048**	**10**	**0.0085**	**0.0099**	**0.0140**	**0.65**
Koza-3	$1 + 4$ CGP	150	–	–	–	7	0.0104	0.0325	0.0328	0.21
	$1 + \lambda$ CGP	120	10	–	16	2	0.0087	0.0099	0.0325	0.49
	$\mu + \lambda$ **CGP**	**80**	**20**	**14**	**4096**	**5**	**0.0091**	**0.0100**	**0.0327**	**0.53**
Nguyen-4	$1 + 4$ CGP	120	–	–	–	10	0.0120	0.0324	0.0487	0.21
	$1 + \lambda$ CGP	40	8	–	64	15	0.0129	0.022	0.0395	0.06
	$\mu + \lambda$ CGP	**60**	**6**	**18**	**2048**	**10**	**0.0101**	**0.0283**	**0.0498**	**0.24**
Nguyen-5	$1 + 4$ CGP	60	–	–	–	7	0.0090	0.0100	0.0240	0.50
	$1 + \lambda$ CGP	150	10	–	16	2	0.0099	0.0099	0.0229	0.50
	$\mu + \lambda$ **CGP**	**150**	**20**	**22**	**4096**	**1**	**0.0085**	**0.0096**	**0.0100**	**0.77**
Nguyen-6	$1 + 4$ CGP	100	–	–	–	2	0.0270	0.0382	0.0392	0.17
	$1 + \lambda$ **CGP**	**60**	**20**	–	**8**	**1**	**0.0091**	**0.0191**	**0.0381**	**0.44**
	$\mu + \lambda$ CGP	80	14	–	4096	5	0.0100	0.0381	0.0407	0.25
Nguyen-7	$1 + 4$ CGP	200	–	–	–	7	0.0157	0.0262	0.0534	0.18
	$1 + \lambda$ **CGP**	**120**	**8**	–	**4096**	**7**	**0.0099**	**0.01866**	**0.0382**	**0.25**
	$\mu + \lambda$ CGP	150	6	2	32	2	0.0116	0.0216	0.0288	0.20
Nguyen-8	$1 + 4$ CGP	150	–	–	–	15	0.0084	0.0111	0.0415	0.53
	$1 + \lambda$ CGP	80	10	–	16	2	0.0072	0.0084	0.0098	0.85
	$\mu + \lambda$ **CGP**	**150**	**6**	**2**	**32**	**2**	**0.0072**	**0.0088**	**0.0095**	**0.98**
Nguyen-9	$1 + 4$ **CGP**	**150**	–	–	–	**15**	**0.2475**	**0.4184**	**1.2077**	**0.00**
	$1 + \lambda$ CGP	200	4	–	16	7	0.2707	0.6189	1.0801	0.01
	$\mu + \lambda$ CGP	120	20	22	4096	15	0.5325	0.7245	1.0079	0.00
Nguyen-10	$1 + 4$ CGP	60	–	–	–	20	0.5728	0.9185	1.1150	0.01
	$1 + \lambda$ CGP	120	10	–	4096	20	0.3718	0.5727	0.7346	0.01
	$\mu + \lambda$ **CGP**	**150**	**20**	**8**	**4096**	**15**	**0.2975**	**0.4020**	**0.5921**	**0.00**
Keijzer-4	$1 + 4$ CGP	22	–	–	–	5	3.6828	3.6828	3.6828	0.00
	$1 + \lambda$ CGP	200	20	–	16	7	2.1038	2.3413	2.4953	0.00
	$\mu + \lambda$ **CGP**	**120**	**20**	**22**	**1024**	**10**	**2.0837**	**2.2254**	**2.3484**	**0.00**
Keijzer-6	$1 + 4$ CGP	100	–	–	–	2	0.3229	0.4883	0.6438	0.00
	$1 + \lambda$ CGP	60	20	–	64	10	0.1538	0.2184	0.3445	0.00
	$\mu + \lambda$ **CGP**	**200**	**20**	**6**	**256**		**0.0516**	**0.1008**	**0.2390**	**0.07**
Pagie-1	$1 + 4$ CGP	150	–	–	–	20	31.5965	34.0846	35.2309	0.00
	$1 + \lambda$ CGP	200	20	–	512	10	14.9535	21.4781	30.7461	0.00
	$\mu + \lambda$ **CGP**	**200**	**20**	**14**	**256**	**15**	**14.7931**	**21.3225**	**30.1226**	**0.00**

$(3,3,1)$- and $(4,4,1)$-adder, $(3,3)$-multiplier, and the parity benchmarks, the best performing algorithm is 1.3 to 3 times faster than the baseline (1+4) CGP. When looking at the CE metric, SA is the clear winner for all but the smallest and the $(3,3,1)$-adder benchmarks. Sometimes, the best performing algorithm regarding the median number of fitness evaluations is not the winner regarding the CE. However, the differences in medians and CE values between the winner algorithm regarding the median and the winner algorithm regarding the CE are small to marginal.

Although we have shown for Boolean benchmarks that the conventional way of parameterizing CGP can always be outperformed, we would like to emphasize the following fact: Neither the best performing algorithm regarding the median nor the best algorithm regarding the CE metric can be in general considered dominant when it comes to the computational complexity of optimization and with it, time. The reason for this is the inaccurate assumption that the computational complexity of a fitness evaluation is constant among all CGP parameterizations. For example, $(\mu+\lambda)$ CGP is the best-performing algorithm regarding the median and CE metrics for the $(2,2,1)$-adder. However, despite a worse median and CE values, (1+4) CGP operating on a single-line CGP and SA evolve functionally correct adders in a much shorter time. This is because the genotype sizes found by iRace are much smaller for the two algorithms than for the $(\mu+\lambda)$ CGP. But even with identical CGP geometries, the functional evaluation complexity can vary greatly, as the number of active genes that are processed by the fitness evaluation procedure can be different. With this observation in mind, we will investigate in future work how the algorithms compete when optimizing for absolute normalized CPU time on single-core and parallel architectures.

The second observation is that when tuning for λ or for λ and μ, small values are identified by iRace as beneficial. With this, HC and its close derivatives seem to work better for CGP when optimizing Boolean circuits.

In related work [99], it was shown that the efficiency of (1+4) CGP on single-line CGP increases with rising n_c. This can also be observed in Table 5.5. However, the efficiency of CGP can be improved using rectangular grids and slightly different $(\mu+\lambda)$ CGP schemes as well as SA. This is our third observation for the evolution of Boolean functions.

5.5.2 Evolution of Symbolic Regression Function

For symbolic expression, we have skipped experiments with SA, as we must investigate more cooling schemes to make SA competitive.

The first observation of Table 5.6 is that the regular (1+4) CGP can always be outperformed regarding approximation accuracy except for the Nguyen-8 benchmark. The second observation is that $(\mu+\lambda)$ CGP is very successful. Except for three benchmarks, it is constantly better than the other algorithms. For the symbolic regression, we cannot observe increased efficiency for single-line CGP when increasing n_c. However, and this is our next observation, the number of offspring individuals is usually very large. This is similar to regular GP, where often large populations are

used.
The last two findings in Table 5.6 are: Similar to Boolean functions, rectangular CGP geometries are more efficient than single-line CGP and successful mutation rates are rather high, which is in contrast to prior findings suggesting to set the mutation rate as low as possible.

5.6 Conclusion

This chapter described an empirical study investigating if the usual way CGP is parametrized in related work is actually the best choice. The results are that, indeed, the single-line CGP with an $(1 + 4)$ CGP scheme is good for Boolean benchmarks but that much better results can be achieved for Boolean and symbolic regression functions when using rectangular CGP grids and differently parametrized $(\mu + \lambda)$ CGP schemes as well as SA. Furthermore, we could observe that similar to GP, CGP greatly benefits from large exploration rates, i.e. large offspring populations and high mutation rates, when evolving symbolic regression functions. This behavior is surprising and requires further investigation. It is especially interesting if the previous results on inner CGP mechanisms, like “neutrality”, are still valid.
The following recommendations can be drawn from our experiments.

- For simple Boolean functions the (1+1)-CGP with 30 to 50 rows, and 100 to 200 columns perform best.
- For complex Boolean functions, SA applied on CGP with 3 to 10 rows, and 30 to 300 columns perform best. Increasing the number of rows to 100 might help in the case of heavy functions, such as multiplication.
- For Boolean functions, the best-observed mutation rate interval is $[0.1, 1.6]\%$.
- For continuous functions, CGP with 3 to 20 rows and 80 to 200 columns performs best.
- For continuous functions CGP with $\mu = 2 \ldots 22$ and $\lambda = 2048 \ldots 4096$ performs best. However, it is worth investigating $\lambda = 8 \ldots 32$ in cases where large λ values do not result in fast convergence.
- For continuous functions, the mutation rate may vary from 1% to 15%, with higher mutation rates being more successful for larger genotypes.

Finally we can conclude, that we have demonstrated that the $(1+4)$-CGP is not the best choice for a set of popular GP benchmark problems in two different problem domains. Our experiments clearly show that the choice of the $(1+4)$-CGP is inferior to other $(\mu + \lambda)$ settings.

6 A Self-adaptive Strategy for CGP

6.1 Introduction

This chapter introduces an adaptive strategy for real-valued CGP, which is based on the population statistics of modern GP systems. Moreover, the chapter shows how these statistics can be used to maintain population diversity. Our experiments show that our strategy improves the convergence rate. The new strategy has been tested on several regression problems. Our strategy benefits from locating suitable points in the evolutionary process to adapt the probabilities of the genetic operators. As a result, population diversity is increased, which may lead to better convergence. On harder problems, the new strategy also benefits from adapting the selection pressure. To locate opportunities for adaption, a new metric for CGP is introduced, which measures the healthy population diversity.

6.2 Related Work

For the real-valued representation of CGP, Clegg et al. [13] demonstrated the problem of low convergence in real-valued CGP on a regression problem and introduced a variable crossover. On the issue of algorithm stagnation of EAs, previous research in genetic algorithms [46] showed that adaptive genetic operators and selection, which are using population statistics, are helpful to maintain population diversity. Many adaptive strategies benefit from adapting the probabilities of crossover and mutation [17, 134, 143]. Later, McGinley et al. [91] proposed an adaptive genetic algorithm that also uses an adaptive selection method by measuring the healthy population diversity. Clegg et al. introduced a variable crossover operator for CGP, though it merely performs a change to mutation only CGP by gradually decreasing the crossover probability.The variable crossover is based on their observation of regression problems and does not solve the problem itself.

6.3 Population Statistics in CGP

In CGP, two types of population statistics can be used. Information about the population can be received by exploring the fitness landscape and the phenotype space. On the problem of premature convergence, many adaptive approaches react on a homogeneous fitness landscape by increasing the influence of the mutation operator to increase population diversity. Besides the classical approach of using population statistics by exploring the fitness landscape, Meier et al. explored the phenotype space for the forking operator [93], a genetic operator which replaces solutions which have a high phenotypic frequency within the population with genotypic variations to achieve more population diversity. The phenotype of every individual can be expressed as a string representation called fingerprint. In CGP, different genotypes can refer to the same phenotype. This property is an example of neutrality in CGP and was outlined by Miller and Thomson [100]. The fingerprint of a phenotype is used to determine its frequency in the population. To accomplish this, the textual representation of the phenotype can be used as the fingerprint. The number of unique phenotypes can be used to classify the diversity of the population. In this way, homogeneous areas in the solution space can be detected. This can be more effective than merely exploring the fitness landscape.

6.4 Adaption of Genetic Operators

Evolutionary algorithms handle with two primary abilities — exploitation of the genetic material in the population and exploration of the search space.In the field of EAs, exploitation is mostly done by crossover, and mutation is used for exploration. Exploitation and exploration can be controlled by the probabilities of crossover and mutation.Let crossover probability be the probability that the crossover occurs when two parents have been selected; otherwise, one of the parents is passed through by random. Let mutation probability the probability that a gene of the genotype is mutated (replaced by a random number). Finding the right probabilities of the genetic operators is the key to an efficient search. Running the algorithm with inappropriate probabilities can lead to premature convergence, algorithm stagnation or excessive diversity. During the evolutionary process, the conditions are changing from one generation to the next so that the probabilities can be adapted to the current circumstances. Previous research showed that increasing crossover probability at high population diversity and increased mutation probability at low diversity has a beneficial effect [91]. The key to success for an adaptive strategy is the estimation of the right moments for adapting the conditions for convergence toward the global optimum or in contrast to explore the solution space further. Besides the adaption of the probabilities of the genetic operators, controlling selection pressure has also shown a beneficial effect [91].

6.5 Introducing the New Strategy

```
o0 = * (- 1.0 (* 1.0 (* x x)))
     (- (* 1.0 (* x x)) (* (* x x) (* x x)))
```

Listing 6.1: Textual representation of an phenotype of fitting the function $x^6 - 2x^4 + x^2$ "

6.5.1 Measuring Phenotype Space Diversity

Measuring the diversity of the phenotype space is the core of our adaptive strategy. Where other strategies measure the diversity only in genotype space, our strategy also benefits from measuring the diversity in phenotype space. Let S be the phenotype space. Individual $s \in S$ with $s = \Sigma^*$ over the alphabet Σ is the textual representation of a certain phenotype. An example of a textual representation of s is shown in Listing 6.1. Let N be the number of individuals and $S^* \subset S$ the subset of phenotypes described by all individuals. Phenotype space diversity θ can be split into two terms, standard phenotype diversity θ^{s} and healthy phenotype diversity θ^{h}. We receive information about the diversity of the whole phenotype space from θ^{s}, where θ^{h} refers to the ratio of healthy phenotypes in the population. A healthy phenotype is of high fitness and unique in the phenotype space, which is of high value for the evolutionary process. To determine θ^{s}, we define a dictionary $M : S \to \mathbb{N}_0$, which shares all fingerprints of the phenotype space. As the value of $M(j)$, the frequency of a phenotype in the population is stored. The value of θ^{s} is calculated by the size of the dictionary in ratio to the number of individuals, as shown in Equation (6.4). When every individual refers to one phenotype, the dictionary size is equal to the population size, which is the best standard diversity in phenotype space. For θ^{h}, we first have to determine the phenotype health of every individual in the population. For calculating the phenotype health, the fitness rate F^{Rate} (Equation (6.2)) of an individual and the frequency of the phenotype in relation to the diversity of the population f^{Rate} (Equation (6.1)) have to be determined. In Equation (6.1), $f^{\mathrm{Rate}}(j)$ stands for the frequency rate of j-th phenotype which is stored in the dictionary M. In Equation (6.2) $F(i)$ stands for the fitness value of i-th individual in the population. Furthermore, F^{Worst} and F^{Best} represent the best and worst fitness value in the population. The frequency rate f^{Rate} is multiplied with an amplifier α as shown in Equation (6.3) to amplify lower frequencies and is limited to a maximum with a value of 1. In the discussion section, we advise the parametrization of the amplifier, which is based on our experiments. Finally, phenotype health can be determined as the product of the fitness rate and negated frequency rate, since lower frequency rates are better. The sum of the phenotype health values over the population can be used to describe θ^{h} and is normalized by N, consider Equation (6.5). To calculate

θ, we average θ^{s} and θ^{h} as shown in Equation (6.6).

$$f^{\mathrm{Rate}}(j) := \frac{M(j)}{|M|}, \quad \text{for all } j \leq |S^*| \tag{6.1}$$

$$F^{\mathrm{Rate}}(i) := \frac{F(i) - F^{\mathrm{Worst}}}{F^{\mathrm{Best}} - F^{\mathrm{Worst}}}, \quad \text{for all } i \leq N \tag{6.2}$$

$$w(j) := \min(\alpha \cdot f^{\mathrm{Rate}}(j), 1), \quad \text{for all } j \leq |S^*|, \alpha \in \mathbb{N} \tag{6.3}$$

$$\theta^{\mathrm{s}} := \frac{|M|}{N} \tag{6.4}$$

$$\theta^{\mathrm{h}} := \frac{1}{N}\sum_{i=1}^{N}(1 - w(i)) \cdot F^{\mathrm{Rate}}(i), \quad \text{for all } i \leq N \tag{6.5}$$

$$\theta := \frac{\theta^{\mathrm{s}} + \theta^{\mathrm{h}}}{2} \tag{6.6}$$

6.5.2 Adapting Crossover Probability

With information about the phenotype space diversity, the crossover probability can be adapted to the current conditions. When θ is high, the population consists of healthy phenotypes and less homogeneous areas. The diversity of the individuals is high, which is a good point to convert the population. In this case-crossover probability is increasing to progress the population toward the global optima. Otherwise, when θ is low, we have to handle with more homogeneous areas where a high crossover probability can convert the population to a local optimum. In this case, crossover probability is reduced to increase diversity through mutation. A lower and upper limit limits the crossover probability within the range $[C_L,C_H] \in \mathbb{R}$ where $0 \leq C_L < C_H \leq 1$. This prevents crossover probability from exceeding boundaries. Equation (6.7) shows the adaption of the crossover probability. The limits C_L and C_H are mostly set empirically. In CGP, the crossover probability often varies between the value $C_L = 0.5$ and $C_H = 1$.

$$p^{\mathrm{Cross}} = \theta \cdot (C^{\mathrm{H}} - C^{\mathrm{L}}) + C^{\mathrm{L}} \tag{6.7}$$

6.5.3 Adapting Mutation Probability

In contrast, to adapt a global crossover probability, mutation probability is adapted locally. Since mutation has a strong effect on the phenotype in CGP, a global adaptive mutation is not recommended. By increasing the mutation probability for all phenotypes, good solutions with high fitness would not be protected by altering their genotype through mutation. Also, the population is becoming too homogeneous when the mutation probability is too low. The mutation probability is adapted directly in phenotype space based on the health of the phenotype. Healthy phenotypes are protected by a lower probability where unhealthy phenotype will be mutated with higher probability. As a result, diversity is increased in areas where the population is too homogeneous. Its frequency can determine the need for a higher mutation rate for a phenotype. The mutation probability is calculated as shown in Equation (6.8) with the limits M^{L} and M^{H}. Since each individual has its mutation probability, the probabilities of two individuals are averaged if crossover occurs. Like the limits of the crossover probability, M^{L} and M^{H} are mostly set empirically. In GGP, the mutation probability often varies between a value of 0.01 and 0.3.

$$p^{\mathrm{Mut}}(i) = w(i) \cdot (M^{\mathrm{H}} - M^{\mathrm{L}}) + M^{\mathrm{L}} \tag{6.8}$$

6.5.4 Adapting Selection Pressure

On more complex problems we also adapt the selection pressure in relation to θ^{h}. We adapt the selection pressure by adjusting the tournament size for the tournament selection method since we use this method in combination with our adaptive strategy. When θ^{h} is high, the selection pressure is increased to select individuals with high fitness. At low θ^{h}, selection pressure is decreased, to give lower fitting individuals a greater chance of being selected. Normally we handle the principle of survival of the fittest, but in situations with a homogeneous population, higher selection pressure is not beneficial. By selecting lower fit individuals who are of higher diversity, θ^{h} is increased. As we use tournament selection, the tournament size is adapted, as shown in Equation (6.9). Like in traditional GP, we tend to use higher selection pressure. The limits T^{H} and T^{L} are set empirically, as we show in the next section.

$$T^{\mathrm{Size}} = \theta^{\mathrm{s}}(T^{\mathrm{H}} - T^{\mathrm{L}}) + T^{\mathrm{L}}, T^{\mathrm{Size}} \in \mathbb{N} \tag{6.9}$$

6.6 Results

In this section, we compare our new strategy with the traditional real-valued CGP [13] on four different regression problems. We perform two different types of experiments. The first experiment is similar to the experiments of Meier et al. [93] with a fixed tournament size to evaluate the use of adaptive crossover and mutation. For the first experiment we chose the regression problems f_1 and f_2. The second experiment was performed on the more complex regression problems f_3 and f_4. For this experiment we also use adaptive selection pressure. Let $D = \{x_p\}_{p=1}^{\mathcal{P}}, x_p \in [-1, 1]$ be a training dataset of $\mathcal{P}$ random points and $f_{\mathrm{ind}}(x_p)$ the value of an evaluated individual and $f_{\mathrm{ref}}(x_p)$ the true function value. Let

$$C := \sum_{p=1}^{\mathcal{P}} |f_{\mathrm{ind}}(x_p) - f_{\mathrm{ref}}(x_p)|$$

be the cost function. When the difference of all absolute values becomes less than 0.01, the algorithm is classified as converged. To evaluate the results, we use a methodology based on the work of Meier et al. [93] and Clegg et al. [13], which includes the average number of generations until convergence and the computational effort. For calculating the computational effort, we take the minimum computational effort (Min. CE), as shown in Equation (5.5). For each average number, e.g. 89±216, the first number refers to the average value and the second to the standard deviation. Furthermore, we classify every run, which exceeds 1000 generations as a slow run and summarized the number of slow runs for each problem. We perform 1000 independent runs with different random seeds on every regression problem and use the Mann-Whitney-U-Test to classify the significance of the results. The average number of generations is denoted with a^* if the significance level is $P < 0.05$ or $a^\dagger$ if the significance level is $P < 0.01$. To illustrate the convergence behavior, the function value of the best solution has been averaged for each generation over all runs.

$$f_1(x) = x^6 - 2x^4 + x^2 \tag{6.10}$$

$$f_2(x, y) = (x^2 \cdot y^2)/(x + y) \tag{6.11}$$

$$f_3(x) = x^5 - 2x^3 + x \tag{6.12}$$

$$f_4(x) = x^4 + x^3 + x^2 + x \tag{6.13}$$

6.6.1 Experiment I

For the first experiment, we use the same parameter configuration as Meier et al. for the traditional real-valued CGP, which seem to be good choices for the probabilities for mutation and crossover on the tested regression problems. The parameter configuration for the traditional real-valued CGP and our adaptive approach is shown in Table 6.1. Our adaptive strategy uses the same configuration except the mutation and crossover probability. For the adaptive probabilities and the amplifier, we define a general configuration for all problems, which is marked as general. This setting is the same for all tested problems and has been determined empirically on the set of benchmark problems. Additionally, we determined a specific configuration for each problem, which is marked as specific and represents the optimal strategy parametrization for the given problem. The reason for the use of a general and specific configuration is to demonstrate the generalization abilities of our strategy as well as the increase of the performance For example, for Problem 1, we use a crossover probability ranging from 0.7 to 1.0, and α is set to 5. The optimal strategy parametrization has been determined empirically. Since our strategy maintains population diversity, we can use higher crossover probabilities with less risk of trapping into local optima.

Table 6.2 shows the results for the first problem, which outlines that the adaptive strategy convergences faster with general and specific settings. Figure 6.1 shows a comparison of the average convergence between the traditional Real-Value CGP

Table 6.1: Algorithm configuration for the first experiment

Property	Traditional	Adaptive
Maximum node count	10	same
Function lookup table	+ (0), - (1), * (2), / (3)	same
Population size	50	same
Maximum Generations	20,000	same
Crossover operator	weighted average	same
Crossover probability (P^{Cross})	0.75	0.5 - 1.0 (general)
Mutation operator	Reset gene $\in [0, 1]$	same
Mutation probability (P^{Mut})	0.2	0.2 - 0.3 (general)
Tournament selection size (T^{Size})	20	same
Elitism size	2	same
Amplifier	-	3 (general)

Table 6.2: The average number of generations and computational effort required by CGP for the problem $f_1(x) = x^6 - 2x^4 + x^2$

Algorithm	Avg. Generations	Min. CE	Slow Runs
Traditional	151 ± 712	22622	17
Adaptive (general)	$101 \pm 169^*$	20380	5
Adaptive (specific)	$89 \pm 128^\dagger$	17828	3

Table 6.3: The average number of generations and computational effort required by CGP for the problem $f_2(x, y) = (x^2 \cdot y^2)/(x + y)$

Algorithm	Avg. Generations	Min. CE	Slow Runs
Traditional	313 ± 612	60689	71
Adaptive (general)	$222 \pm 444^\dagger$	44559	31
Adaptive (specific)	$214 \pm 422^\dagger$	42915	33

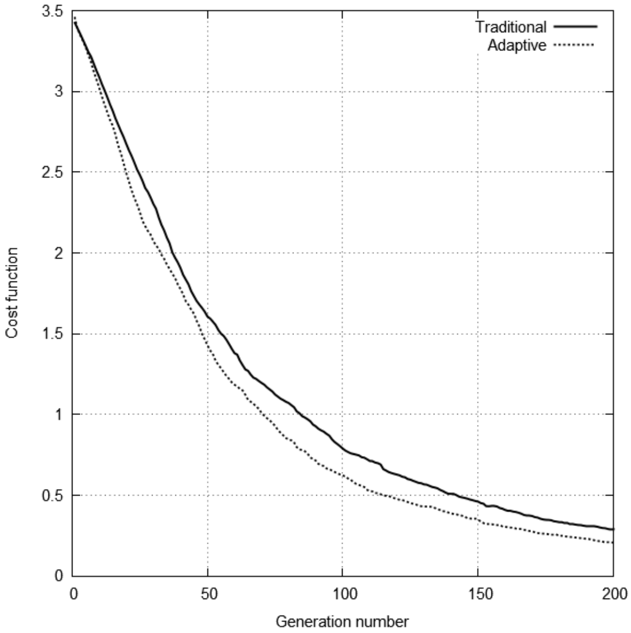


Figure 6.1: Average convergence for the first generations covering traditional Real-Value CGP and adaptive Real-Value CGP with specific settings on $f_1(x) = x^6 - 2x^4 + x^2$.

and our adaptive approach with specific settings for the former generations. Our adaptive method converges faster and reaches the convergence criteria faster, as illustrated in Figure 6.2. Table 6.3 shows the result of the second regression problem. As illustrated, the number of generations until convergence and the computational effort is better in comparison to the traditional Real-Value-CGP. Figure 6.3 and 6.4 underline the faster convergence of our adaptive approach.

6.6.2 Experiment II

For the second experiment, we use the same algorithm configuration as shown in Table 6.1, except for the tournament size. The configuration of the tournament size is shown in Table 6.4. For the problems f_3 (Equation 6.12) and f_4 (Equation 6.13) we also use the general and specific set of probabilities as we did in our first experiment. For this experiment, we use the plot style, as shown in Figure 6.5, because the presented problems use high polynomials, which produce high fitness values. As a result, the detailed convergence for the given problems is difficult to plot. Table 6.5 shows the result for the third regression problem, and as illustrated, the average generation number and the computational effort is also better. The adaptive selection pressure helps to maintain population diversity but also in processing the population toward the global optimum by adjusting high selection pressure at the right time. Table 6.6 shows the result for the last regression problem, which also shows a better

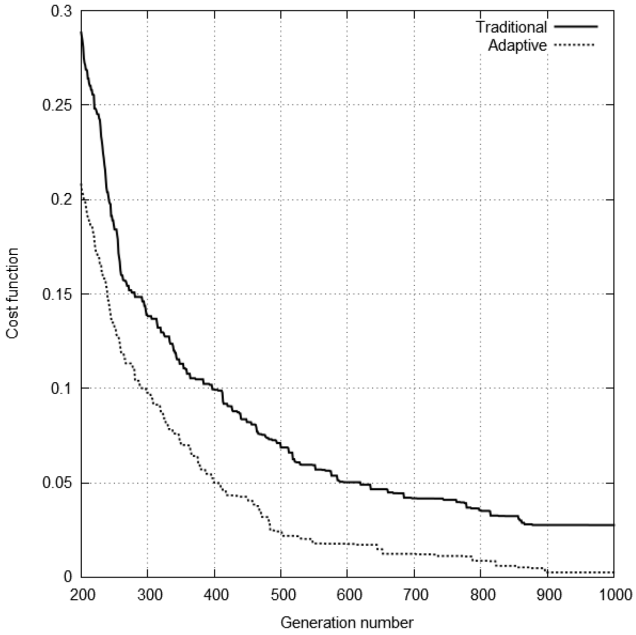


Figure 6.2: Average convergence for the latter generations covering traditional Real-Value CGP and adaptive Real-Value CGP with specific settings on $f_1(x) = x^6 - 2x^4 + x^2$.

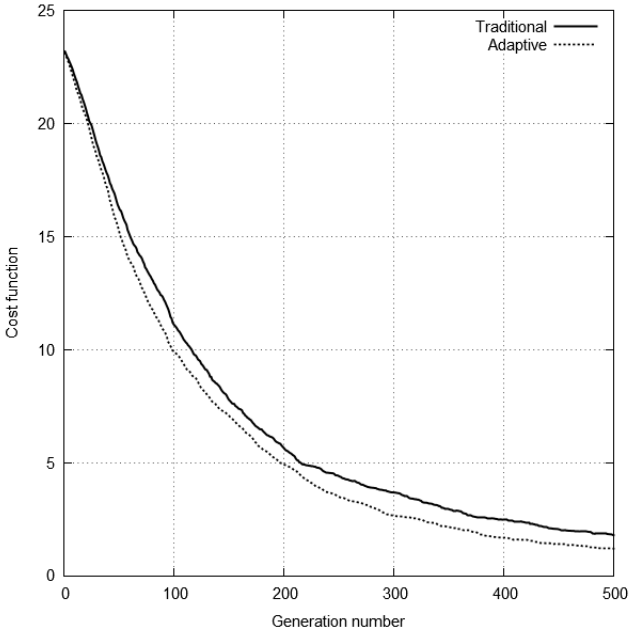


Figure 6.3: Average convergence for the first generations covering traditional Real-Value CGP and adaptive Real-Value CGP with specific settings on $f_2(x, y) = (x^2 \cdot y^2)/(x + y)$.

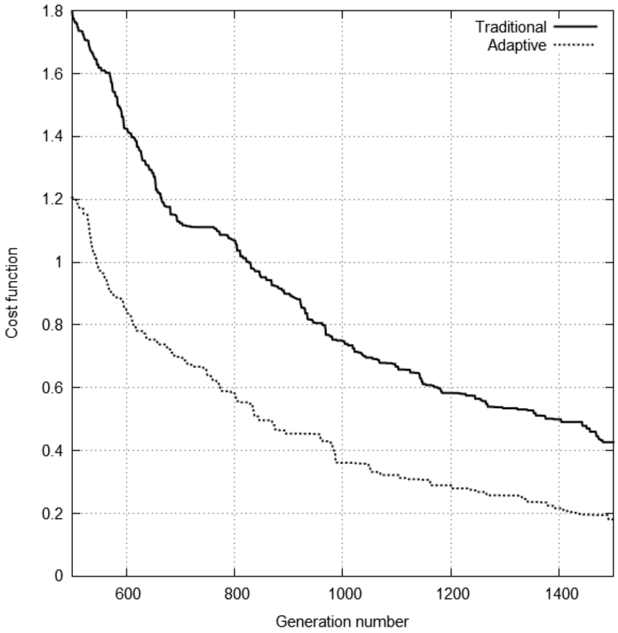


Figure 6.4: Average convergence for the latter generations covering traditional Real-Value CGP and adaptive Real-Value CGP with specific settings on $f_2(x, y) = (x^2 \cdot y^2)/(x + y)$.

Table 6.4: Tournament size configuration for the second experiment

Algorithm	Tournament Size (T^{Size})
Traditional	8
Adaptive	4-10

outcome for our adaptive approach. Also, the number of generations to convergence is better, as shown in Figure 6.6. Our adaptive approach has also slow runs but prevents the occurrence of prolonged runs.

6.6.3 Diversity Comparison

Since $f_4(x)$ is the most complex problem of our test suite which we used for our experiments, we choose this problem for a diversity comparison in phenotype space. We average the diversity value θ (Equation 6.6) of 100 runs which exceed 500 generations for each generation. Based on our second experiment, we use adaptive genetic operators and selection pressure for the adaptive strategy. Since the diversity value θ is normalized and ranges between 0 and 1, a value of 1 represents a phenotype space where every phenotype is unique. The diversity comparison is shown in Figure (6.7), and as shown, the average diversity value of the adaptive algorithm is much higher.

Table 6.5: The average number of generations and computational effort required by CGP for the problem $f_3(x) = x^5 - 2x^3 + x$

Algorithm	Avg. Generations	Min. CE	Slow Runs
Traditional	524 ± 1005	104965	126
Adaptive (general)	$351 \pm 564^{\dagger}$	70337	70
Adaptive (specific)	$349 \pm 551^{\dagger}$	69892	67

Table 6.6: The average number of generations and computational effort required by CGP for the problem $f_4(x) = x^4 + x^3 + x^2 + x$

Algorithm	Avg. Generations	Min. CE	Slow Runs
Traditional	1009 ± 1425	252378	328
Adaptive (general)	$713 \pm 909^{*}$	178296	265
Adaptive (specific)	$620 \pm 820^{\dagger}$	155246	217

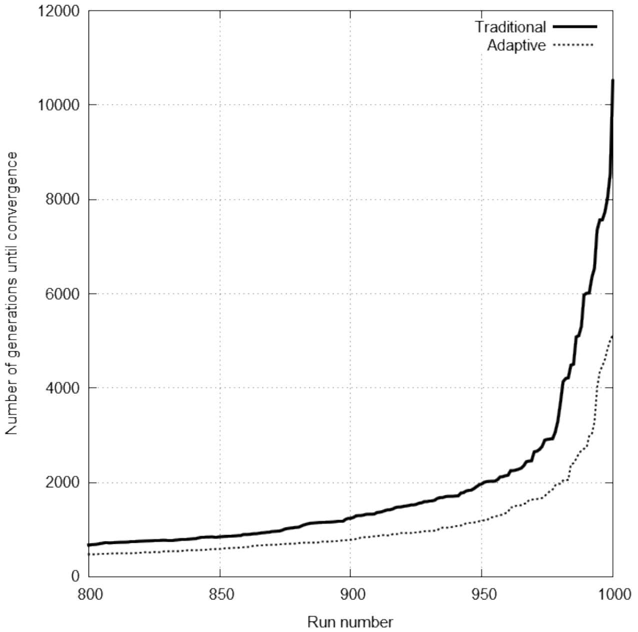


Figure 6.5: The number of generations to convergence over the slowest 200 runs for traditional Real-Value CGP and adaptive Real-Value CGP with specific settings on $f_3(x) = x^5 - 2x^3 + x$.

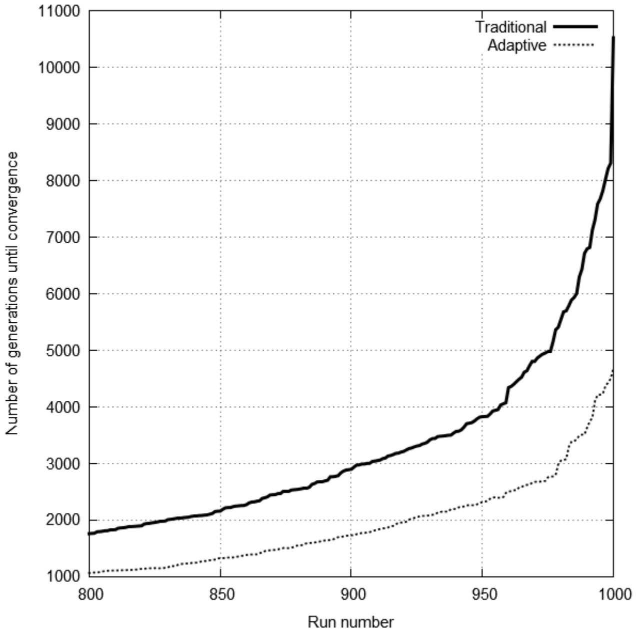


Figure 6.6: The number of generations to convergence over the slowest 200 runs for traditional Real-Value CGP and adaptive Real-Value CGP with specific settings on $f_4(x) = x^4 + x^3 + x^2 + x$.

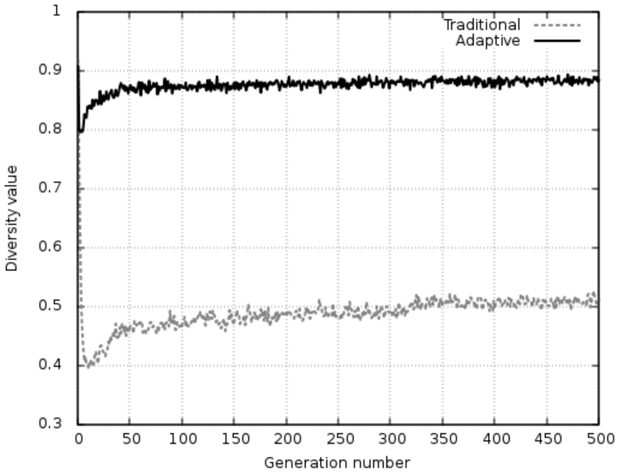


Figure 6.7: Phenotype space diversity comparison for traditional Real-Value CGP and adaptive Real-Value CGP with specific settings on $f_4(x) = x^4 + x^3 + x^2 + x$.

6.7 Discussion

Our experiments on the four regression problems show that using adaptive genetic operators and selection could be beneficial for real-valued CGP. By maintaining population diversity, the occurrence of algorithm stagnation could be prevented. As shown in our first experiment, our strategy improves convergence in the former and latter generations by maintaining population diversity. In the first generations, our strategy benefits from the ability to use higher probabilities of crossover. Using high crossover probabilities with traditional CGP may increase the risk of the occurrence of algorithm stagnation. The population converges to local optima, and the effect of mutation for exploration of the search space is too low. In this case, our new strategy reacts by increasing the impact of mutation, which prevents the populations from getting too homogeneous. The behavior of the adaption process for the mutation and crossover probability is illustrated and stylized in Figure 6.8. It is visible that the mutation probability increased and the crossover probability is simultaneously decreased. The use of a general set of parameters showed the flexibility of our new strategy when the optimal configuration is not set. Our adaptive strategy uses one new parameter α and the lower and upper bounds for the respective intervals, which depends on the complexity of the problem and size of the population. For the choice of the amplifier value α, we have found no general approach yet. Since this value was chosen empirically, we observed that a higher value for the harder problems of our tested regression problems was beneficial. Since modern evolutionary computation systems produce population statistics which can be used for our strategy, the determination of the phenotype space diversity becomes easier. Since our strategy operates with two types of population statistics, the use of our strategy with a GP system which has no built-in statistic module is difficult. However, our analysis of the most used evolutionary computation systems showed that nearly every modern system produces statistics. This also includes fingerprints for GP. Since most GP systems provide fingerprints for tree representation, the use of our new strategy in traditional GP is feasible. For selection, we only used the fitness value as a selection criterion. In our experiments, we showed that the crossover of unique and high fitted individuals is beneficial for CGP. For better selection in our strategy, an improved selection method is necessary, which works with two selection criteria, fitness and the frequency of the phenotype in the population. Using the textual representation as a fingerprint is a simple form to map the phenotype space but has the disadvantage that different textual phenotypes are similar in their function. This behavior of neutrality in phenotype space must be investigated in detail to develop more precise mapping techniques. In our experiments, we only used four possible functions and ten nodes per CGP program, but Miller and Smith showed that increasing the number of function node improves the search performance, too [99].

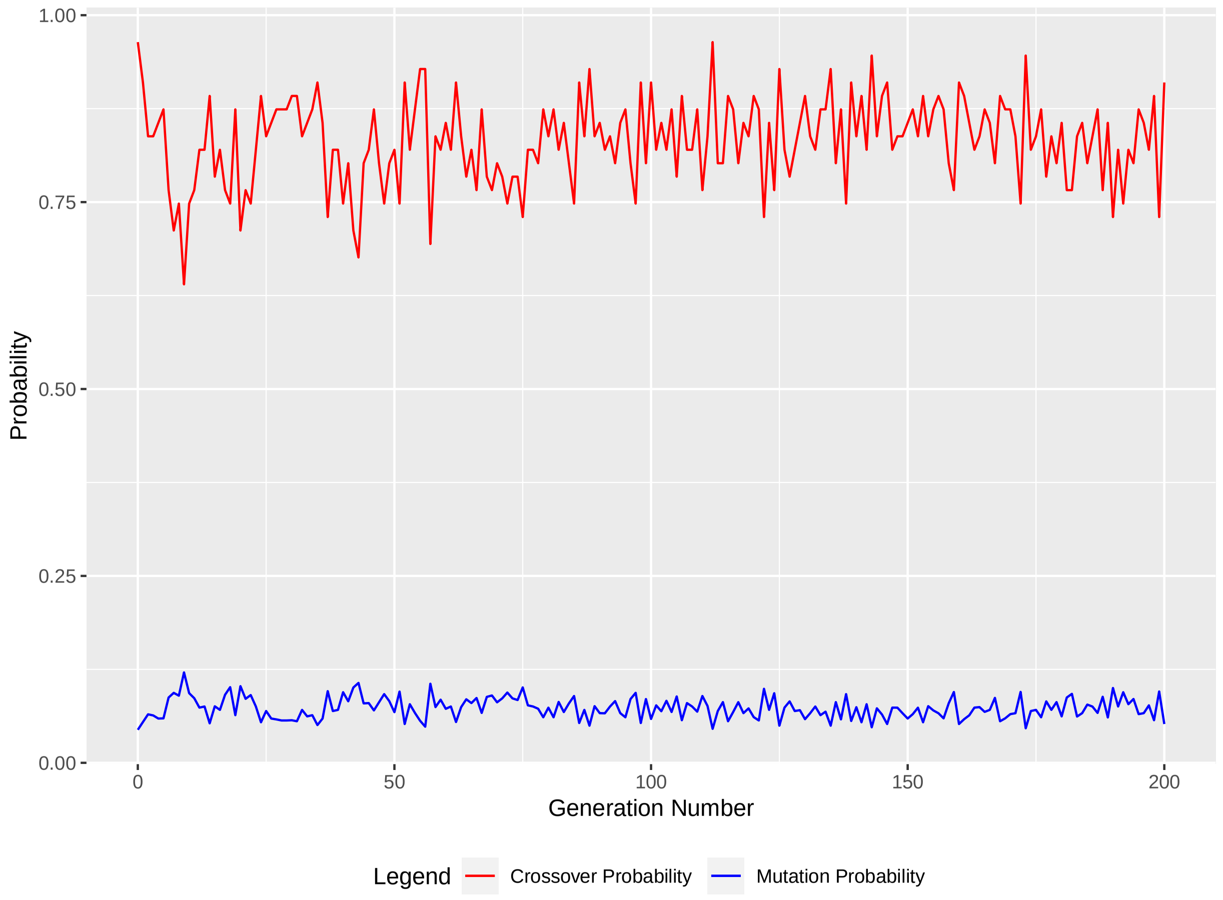


Figure 6.8: Stylized behaviour of the adaption of crossover and mutation probability over 200 generations for the regression problem $f_1(x) = x^6 - 2x^4 + x^2$.

6.8 Conclusions

A first adaptive strategy for CGP has been proposed. Our strategy maintains population diversity by adapting the probabilities of the genetic operators and selection pressure based on measurements in phenotype space. Also, our new metric for measuring the ratio of healthy phenotypes helps to determine opportunities for the adaption. Our strategy works similar to the adaptive strategy of McGinley et al. [91] by adapting crossover probability globally and mutation probability locally. Since some adaptive schemes handle with a global mutation probability, this approach showed no beneficial effect in our experiments. Also increasing mutation and crossover probability simultaneously if the population diversity becomes less as proposed by Srinivas et al. [143], this behavior influenced the convergence in our experiments negatively.

Increasing mutation probability in homogeneous areas of the search space and reducing crossover probability if the population diversity is low as proposed by McGinley et al. [91] was the most successful approach to adapt the genetic operators. It has been shown that mapping phenotype space is beneficial for CGP on several symbolic regression problems so that future work should focus on the improvement of these measurement techniques. Choosing the textual representation as a fingerprint of a CGP program is the simplest form of mapping phenotype space. To improve the measurement of phenotype space, more detailed methods must be found which can handle phenotype neutrality, which means that phenotypes different in their textual representation are semantically equal.

7 Advanced Crossover Operators

7.1 Introduction

Tree-based GP was originally introduced with a crossover technique , which swaps randomly chosen sub-branches of the parent trees to produce new offspring. Koza considered crossover as the dominant genetic operator because of his experiments [67, 68]. However, later research with more comprehensive and detailed experiments found that the beneficial effects of crossover cannot be generalized in GP [86, 87, 161] . In contrast to comprehensive knowledge about crossover in tree-based GP, the state of knowledge in CGP appears to be ambiguous and ambivalent. Furthermore, the potential and understanding of crossover in CGP seem to be an open and remaining question. In this chapter, we introduce two new methods for crossover in CGP.
The proposed crossover techniques are evaluated on symbolic regression, Boolean function, and image operator design problems. In this chapter, we also present the results of a comparative study on crossover in CGP , which includes the comparison of different crossover techniques to the $1 + \lambda$ strategy.

7.2 Previous Work on Crossover in CGP

First attempts of crossover in CGP included four variations of crossover, which were tested on the simple regression problem $x^2 + 2x + 1$. Clegg et al. [13] reported that all four techniques failed to improve the convergence of CGP. Compared to running CGP with mutation only, the addition of these crossover techniques hindered the performance. The four methods were tested on the standard integer-based representation of CGP. For instance, the genetic material was recombined by swapping parts of the genotypes of the parent individuals or randomly exchanging selected nodes. Clegg et al. [13] reported that merely swapping the integers (in whatever manner) in the CGP representation disrupts the performance.
This was the motivation for the introduction of a real-valued representation and new crossover technique for CGP by Clegg et al. [13]. The real-valued representation of CGP represents the directed graph as a fixed length list of real-valued numbers in the interval [0,1]. The genes are decoded to the integer-based representation by their normalization values (e.g. number of functions or maximum input range). The recombination of two genotypes is performed by an arithmetic crossover with a random weighting factor, which can also be found in the field of real-valued GAs. Clegg et al. showed that the new representation, in combination with crossover improves the convergence behavior of CGP. However, for the later generations, Clegg et al.

demonstrated that the use of crossover in real-valued CGP disrupts the convergence on one of the two tested problems. The improved convergence of the arithmetic crossover was evaluated in the domain of symbolic regression and has been found useful in this problem domain [13]. Later work by Turner [147] presented results on three additional classes of computational problems, digital circuit synthesis , function optimization and agent-based wall avoidance. On these problems, it was found that the real-valued representation together with the crossover operation performed worse than standard CGP.

Slaný et al. [138] analyzed the fitness landscapes of functional-level CGP on image operator design problems, including single and multipoint crossover operators. It was demonstrated that the mutation operator and the single-point crossover operator generate the smoothest landscapes for the tested problems.

For a multi-chromosome approach to CGP, Walker et al. [158] investigated a multi-chromosome crossover operator which joins the best chromosome parts from all individuals. This crossover technique was found useful for problems with multiple outputs and independent fitness assignments.

Another positive effect of crossover in CGP was obtained by the use of an implicit context representation for CGP in which recombination is useful for the Even Parity-3 problem [10].

CGP has been extended for the automatic definition and reuse of functions by Walker et al. [155] and Kaufmann et al. [62]. Kaufmann et al. adopted the module creation mechanisms for a cone- and age-based CGP crossover [62]. Cone-based crossover showed good results for functions with repetitive inner patterns, while age-based crossover excels for randomized inner structures.

To our best knowledge, the arithmetic crossover seems to be the only approach that aims at standard CGP and has been able to demonstrate a better convergence if two chromosomes are recombined directly. However, for the harder problem of the two tested symbolic regression problems, Clegg et al. concluded that the arithmetic crossover technique does not have a good effect on convergence. Furthermore, it was found that the arithmetic crossover causes occasional runs with a vast number of generations to converge. This has been the motivation for developing a new crossover technique.

7.3 The Subgraph Crossover

The proposed subgraph crossover for CGP is inspired by the subtree crossover found in tree-based GP. However, a directed acyclic graph enables more connections between the nodes, merely choosing one crossover point within the graph is not sufficient. Furthermore, to recombine subgraphs by just swapping parts of the genotype would be a disastrous approach according to the reportings by Clegg et al. Our approach to recombine two directed acyclic graphs is performed by respecting the CGP phenotype. The phenotype of each individual is represented by the active path of the graph and is determined through the evaluation process. Furthermore, the active path of a graph leads to the semantic value of a certain individual in CGP. As a consequence, we exclusively want to recombine the genetic material of the active paths.

For describing the subgraph crossover procedure, let N_{inputs} be the predefined number of input nodes. In CGP, the nodes are indexed from 0 to $N - 1$ and the input nodes of each graph are indexed from 0 to $N_{\text{inputs}} - 1$. The nodes which lie between the input and output nodes are denoted as function nodes. The crossover is done with two parents which are denoted as P_1 and P_2. For the crossover procedure, the node numbers of the active function nodes are necessary. The node numbers of the active nodes of P_1 and P_2 are stored in two arrays M_1 and M_2. The active nodes are determined by the backward search in the evaluation procedure.

To define one suitable crossover point, we define two possible crossover points C_{P1} and C_{P2} of the two parents. With information about the active nodes and the length of the path, we can choose two possible crossover points. The possible crossover points C_{P1} and C_{P2} are chosen by chance in the range of the active function nodes which are stored in M_1 and M_2. The possible crossover points may not be input or output nodes. A defintion of the subgraph crossover is given in Definition 7.1.

Definition 7.1 (Subgraph crossover)**.** *The subgraph crossover is a variation operator for CGP which exchanges and links subgraphs of function nodes between two selected parents, producing one or two offspring.*

The crossover procedure is done by performing the following steps:

1. **Define a general crossover point**

   A general crossover point C_P is defined by choosing the smaller crossover point from C_{P1} and C_{P2}. The reason for this is that the subgraphs of the parents, which will be placed in front of or behind the crossover point of the offspring's genome should be balanced. The representation of CGP allows active paths of an individual, which can start in the middle or back of the graph. The subgraph which will be placed in front of the crossover point must start at more leading active nodes. If C_P is defined as the possible point C_{P1}, the subgraph of P_1 in front of C_P will be placed in front of C_P in the offspring genome. The subgraph behind C_P of P_2 will be placed behind C_P in the offspring genome

2. **Copy the genetic material in front of the crossover point**

The genetic material for the section in front of C_P is copied from the parent P_1 to the offspring genome. This includes the function nodes from the start of the genome of P_1 until the function node given by C_P. Since the inactive nodes are also genetic material that can become active, we also copy these nodes for further genetic variation steps.

3. **Copy the genetic material behind the crossover point**

   The genetic material for the section behind the crossover point of the offspring genome is copied from the parent P_2 starting at the crossover point until the output. The resulting subgraph S_2 including the output is copied to the offspring genome behind the crossover point. However, the active nodes of the section behind the crossover point can alter the active path in front of the crossover point by referring to inactive nodes. Further steps are necessary to connect both sections.

4. **Connect both sections of the offspring genome**

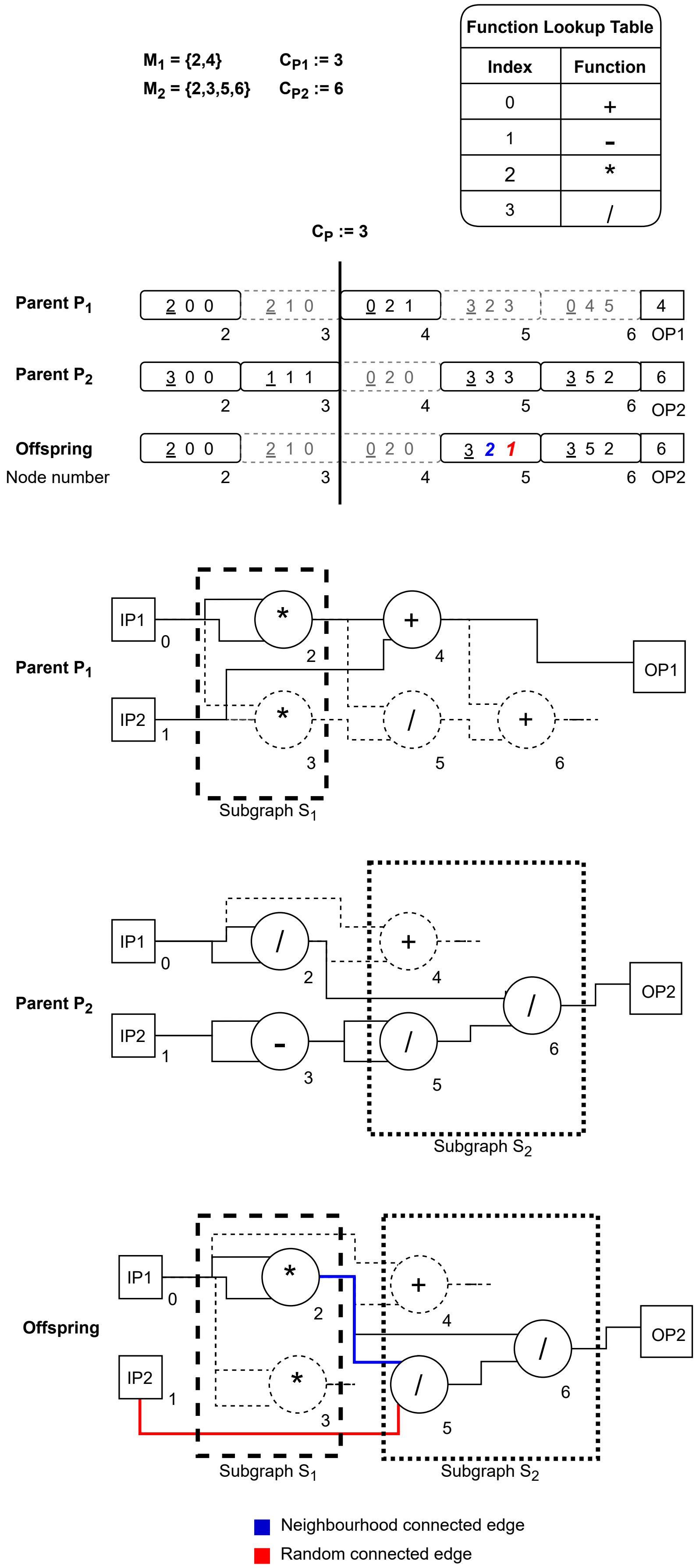
   a) Both sections are connected with a special step that we call *neighbourhood connect*. This step refers to the first active node of the section behind the crossover point which is connected to the last active node of the section in front of the crossover point. This is done by adjusting the connection gene of the first active node of the section behind the crossover point.

   b) To ensure that active nodes of the section behind the crossover point do not refer to inactive nodes of the section in front of the crossover point, we perform a step which we call *random active connect*. All connection genes of the active nodes of the section behind the crossover point are adjusted to the active nodes of the section in front of the crossover point, previous active nodes of S_2 or input nodes. The nodes which are suitable for a random active connection are named as *permissible nodes*. The connection is done by changing the connection gene of a node which refers to an inactive node to a randomly chosen *permissible node*.

The crossover procedure produces a new genome, which represents the offspring concerning the phenotypes of both parents. In the case that two children should be produced, the crossover procedure is performed twice with two different general crossover points. Since the representation of CGP provides connections to any previous function node of the graph, performing only the *neighbourhood connect* could result in a monotone data flow of the resulting phenotype.

The crossover procedure is illustrated in Figure 7.1. At the top of the figure, the arrays with the active nodes and crossover points are listed. Below this information, the genotypes are shown. The figure also shows the phenotypes of the parents and the offspring, the selected subgraphs are marked with dashed boxes.

Algorithms 7.2, 7.3, 7.4 and 7.5 provide pseudo code of the core elements of the subgraph crossover technique. The function *RandomNodeNumber* in Algorithm 7.1 determines a random node number from given lists. The function *DetermineCrossoverPoint* in Algorithm 7.2 calculates and returns the general crossover point and the function *SubgraphCrossover* in Algorithm 7.3 executes the recombination process. The functions *NeighbourhoodConnect* and *RandomActiveConnect* in Algorithm 7.4 and 7.5 handle the rewiring process between the nodes.



Index	Function
0	+
1	-
2	*
3	/

Figure 7.1: The subgraph crossover procedure

Algorithm 7.1 Function for the determination of a random node number

```
    Arguments
    $I$: List with input nodes, null by default
    $N_{\mathrm{F}}$: Sorted list of function nodes. null by default
    $m$: Upper node number limit, null by default
    $n_{\mathrm{i}}$: Number of inputs
    Return
    $N_{\mathrm{R}}[r]$: Selected random node number

function RandomNodeNumber($I$ = null, $N_{\mathrm{F}}$ = null, $m$ = null, $n_{\mathrm{i}}$)
    ▷ Initialize an empty list to store random input and function node numbers
    $N_R \leftarrow$EmptyList()
    if $N_{\mathrm{F}} \neq$ null then          ▷ Check if function node numbers have been passed as argument
        if $m \neq$ null then          ▷ Check if a node number limit has been passed to the function
            ▷ Determine a sublist of $N_{\mathrm{F}}$ where the list elements $X$ of $N_{\mathrm{F}}$ are less or equal $m$
            $N_m \leftarrow N_{\mathrm{F}}$.sublist($X \leq m$)
            if $|N_m| = 0$ then     ▷ If the sublist is emtpy, there are no function nodes before $m$
                $i \leftarrow$ RandomInteger(0, $n_{\mathrm{i}}$)          ▷ Determine a random input node
                $N_R$.append($i$)          ▷ Append the random input to the list
            else
                ▷ Generate a random integer in the range from 0 to $N_m| - 1$ inclusive
                $i \leftarrow$ RandomInteger(0, $|N_m| - 1$)
                $N_R$.append($N_m[i]$)     ▷ Use $i$ as index and get the node number from $N_{\mathrm{F}}$
            end if
        else
            ▷ Otherwise, randomly select a node number in the range from 0 to $|N_{\mathrm{F}}| - 1$ inclusive
            $i \leftarrow$ RandomInteger(0, $|N_{\mathrm{F}}| - 1$)
            $N_R$.append($N_{\mathrm{F}}[i]$)          ▷ Use $i$ as index and get the node number from $N_{\mathrm{F}}$
        end if
    end if
    if $I \neq$ null then          ▷ If the input node are passed to the function
        ▷ Select a input node number in the range from 0 to $|I| - 1$ inclusive by chance
        $j \leftarrow$ RandomInteger(0,$|I| - 1$)
        $N_R$.append($j$)          ▷ Append it to the possible random nodes
    end if
    ▷ Select one node number from the list $N_{\mathrm{R}}$ by chance
    $r \leftarrow$ RandomInteger(0, $|N_{\mathrm{R}}| - 1$)
    return $N_R[r]$          ▷ Return a random input or function node number
end function
```

Algorithm 7.2 Subgraph crossover procedure: Determination of the crossover point

```
    Arguments
    $P_1$: Genome of the first parent
    $P_2$: Genome of the second parent
    $M_1$: List of active nodes of the first parent
    $M_2$: List of active nodes of the second parent
    Return
    $C_\mathrm{P}$: Determined crossover point

function DetermineCrossoverPoint($P_1, P_2, M_1, M_2$)

    $a \leftarrow \min(M_1)$                    ▷ Determine the minimum node number of $M_1$
    $b \leftarrow \max(M_1)$                    ▷ Determine the maximum node number of $M_1$
    ▷ Choose the first possible crossover point by chance in the range from $a$ to $b$ inclusive
    $C_{P1} \leftarrow$ RandomInteger($a, b$)   ▷ Choose the first possible crossover point by chance

    $a \leftarrow \min(M_2)$                    ▷ Determine the minimum node number of $M_2$
    $b \leftarrow \max(M_2)$                    ▷ Determine the maximum node number of $M_2$
    ▷ Choose the second possible crossover point by chance in the range from $a$ to $b$ inclusive
    $C_{P2} \leftarrow$ RandomInteger($a, b$)

    $C_P \leftarrow \min(C_{P1}, C_{P2})$       ▷ The crossover point is the minimum of the possible crossover points
    return $C_P$
end function
```

Algorithm 7.3 Subgraph crossover procedure: Recombination of the genomes

```
Arguments
$P_1$: Genome of the first parent
$P_2$: Genome of the second parent
$n_\mathrm{I}$: Number of inputs
Return
$G_\mathrm{o}$: Genome of the offspring

function SubgraphCrossover($P_1$,$P_2$,$n_\mathrm{I}$)
    $G_1 \leftarrow$ Genome($P_1$)    ▷ Store the genome of parent $P_1$ in $G_1$
    $G_2 \leftarrow$ Genome($P_2$)    ▷ Store the genome of parent $P_2$ in $G_2$

    $M_1 \leftarrow$ DetermineActiveNodes($P_1$)    ▷ Store the active nodes of parent $P_1$ in $M_1$
    $M_2 \leftarrow$ DetermineActiveNodes($P_2$)    ▷ Store the active nodes of parent $P_2$ in $M_2$

    $n_\mathrm{g} \leftarrow |G_1|$    ▷ Determine the number of genes

    $C_P \leftarrow$ DetermineCrossoverPoint($P_1, P_2$)    ▷ Determine the crossover point (Alg. 7.2)

    $p_c \leftarrow$NodePosition($C_P + 1$)    ▷ Determine the crossover point position in the genome
    $G_o[0, p_c - 1] \leftarrow G_1[0, p_c - 1]$    ▷ Copy the part before the crossover position from $G_1$ to $G_o$
    $G_o[p_c, n_g] \leftarrow G_2[p_c, n_g]$    ▷ Copy the part behind the crossover position from $G_2$ to $G_o$

    ▷ Create the list of active function nodes of the offspring

    ▷ Determine and store a sublist of $M_1$ where the list elements $X$ of $M_1$ are less or equal $C_\mathrm{P}$
    $N_{A1} \leftarrow M_1$.sublist($X \leq C_\mathrm{P}$)

    ▷ Determine and store a sublist of $M_2$ where the list elements $X$ of $M_2$ are greater than $C_P$
    $N_{A2} \leftarrow M_2$.sublist($X > C_P$)

    if $|N_{A1}| > 0$ and $|N_{A2} > 0|$ then    ▷ Check if both lists contain active function node numbers
        ▷ Determine the first active node number before the crossover point $C_P$
        $n_F \leftarrow N_{\mathrm{A}1}$.last()
        ▷ Determine the first active node number behind the crossover point $C_P$
        $n_B \leftarrow N_{\mathrm{A}2}$.first()

        $G_o \leftarrow$ NeighbourhoodConnect($n_F$,$n_B$,$G_o$)    ▷ Neighbourhood connect (Alg. 7.4)
    end if

    $N_A$.append($N_{\mathrm{A}1}$)    ▷ Append the sublist $N_{\mathrm{A}1}$ to $N_\mathrm{A}$
    $N_A$.append($N_{\mathrm{A}2}$)    ▷ Append the sublist $N_{\mathrm{A}2}$ to $N_\mathrm{A}$

    if $|N_A| > 0$ then    ▷ Check if any function nodes are active
        $G_o \leftarrow$ RandomActiveConnect($n_\mathrm{I}$,$N_A$, $C_P$,$G_o$)    ▷ Random active connect (Alg. 7.5)
    end if

    return $G_o$
end function
```

Algorithm 7.4 Subgraph crossover procedure: Connecting two recombined subgraphs with the *NeighbourhoodConnect* step

```
Arguments
$n_{\mathrm{F}}$ : Number of the first active node before the crossover point
$n_{\mathrm{B}}$ : Number of the first active node behind the crossover point
$G_{\mathrm{o}}$ : Genome of the offspring
Return
$G_{\mathrm{o}}$ : Genome of the offspring

function NeighbourhoodConnect($n_{\mathrm{F}}$, $n_{\mathrm{B}}$, $G_{\mathrm{o}}$)
    $G_C \leftarrow$ GetConnectionGenes($n_{\mathrm{B}}$, $G_{\mathrm{o}}$)    ▷ Store the connection genes of $n_{\mathrm{B}}$ in $G_{\mathrm{C}}$
    $G_C[0] \leftarrow n_{\mathrm{F}}$    ▷ Adjust the first connection gene of node $n_{\mathrm{B}}$ to the node number $n_{\mathrm{F}}$
    $G_{\mathrm{o}} \leftarrow$ SetConnectionGenes($G_C$,$n_{\mathrm{B}}$, $G_{\mathrm{o}}$)    ▷ Put the modified gene back to $G_{\mathrm{o}}$
    return $G_{\mathrm{o}}$
end function
```

Multiple Outputs

The proposed subgraph crossover primarily focuses on single output problems, but we also investigate multiple output problems in this chapter. On this kind of problems, we deal with multiple active paths that share one genotype. The proposed step *neighbourhood connect* connects two nodes for one active path. This procedure becomes more complicated if numerous active paths are involved. Therefore, on multiple output genotypes, we only connect the two parts of the parent genotypes by performing the *random active connect*. This procedure is also applied to all output nodes since they can refer to inactive nodes in the newly produced genotype.

7.4 The Block Crossover

The block crossover technique focuses on the one-dimensional representation of CGP where the number of rows is limited to one. The block crossover basically swaps the function genes between blocks of active function nodes which have been selected from each parent. Given the previously selected genotypes of two individuals serving as parents, the block crossover first determines a list of the active nodes by evaluating the genotype's active path. Afterward, the function nodes blocks are constructed with respect to the a predifned maximum block size B. The construction is done by randomly selecting a subset of active function nodes from each active function list.

Algorithm 7.5 Subgraph crossover procedure: Connecting two recombined subgraphs with the *RandomActiveConnect* step

Arguments
n_{I} : Number of inputs
N_{A} : List of active function nodes
C_{P} : The crossover point
G_{o} : Genome of the offspring
Return
G_{o} : Genome of the offspring

```
function RandomActiveConnect($n_{\mathrm{I}}$,$N_{\mathrm{A}}$, $C_{\mathrm{P}}$,$G_{\mathrm{o}}$)
    $I \leftarrow$ InputNodes($G_o$)                                ▷ Get the input nodes $G_o$
    for each $n \in N_{\mathrm{A}}$ do                               ▷ Iterate over the active nodes
        ▷ If the node number is greater than the crossover point
        if $n > C_{\mathrm{P}}$ then
            $G_{\mathrm{C}} \leftarrow$ GetConnectionGenes($n$)      ▷ Get the connection genes of the node
            for each $g_c \in G_{\mathrm{C}}$ do                     ▷ Iterate over the connection genes
                ▷ If the current connection gene is not connected to an active function node
                if not $N_{\mathrm{A}}$.contains($g_{\mathrm{c}}$) then
                    ▷ Replace the connection with a random active function node which is located before the crossover point or an input node(Alg. 7.1)
                    $g_c \leftarrow$ RandomNodeNumber($I, N_{\mathrm{A}}, C_{\mathrm{P}}, n_{\mathrm{I}}$)
                end if
            end for
        end if
    end for
    ▷ Adjust the output genes which are currently linked to passive nodes to active function nodes
    $O \leftarrow$ OutputGenes($G_o$)                               ▷ Get the output genes of $G_o$
    for each $g_o \in O$ do                                          ▷ Iterate over the output genes
        if not $N_{\mathrm{A}}$.contains($g_{\mathrm{o}}$) then     ▷ If an output is connected to an inactive node
            $g_o \leftarrow$ RandomNodeNumber($I$,$N_{\mathrm{A}}$)  ▷ Rewire the output to an input or active function node
        end if
    end for
    return $G_o$
end function
```

These random subsets of active function nodes serve as the blocks and are used for the recombination procedure. The procedure ensures that the blocks have the same size. The block crossover then iterates over the blocks and exchanges the function gene between all function nodes which are stored in the blocks. Since the blocks of the two selected parents have the same dimension, the function genes of the nodes are swapped pairwise at each position.
First the active paths are determined and are stored in the lists M_1 and M_2. When at least one parent has no active function nodes, the crossover procedure is aborted. Otherwise, two blocks N_1 and N_2 containing randomly selected active function nodes from their respective lists are determined, This is done with respect to the maximum block size. If at least one parent has less active function nodes then the predefined maximum block size, the minimum size of active nodes is determined between the parents. Afterward, the minimum is taken as block size. To produce the first offspring O_1, the first parent P_1 is cloned, and the function genes of the nodes inside the block N_1, are replaced by the function genes of the nodes from block N_2. The second offspring O_2 is produced in the same way but instead of P_1 and N_2, the second parent $P2$ and the block N_1 are used. Figure 7.2 illustrates the crossover procedure. At the top of the figure, the active function nodes of the parents and the determined node block are listed. These listings are followed by the genotypes of the parents and the offspring. The function genes which have been swapped are highlighted in red and blue colors. Below the listing of the genotypes, the corresponding phenotypes are shown. The function nodes, which have been altered by the swapping process are also highlighted in red and blue colors. A defintion of the block crossover is given in Definition 7.2.

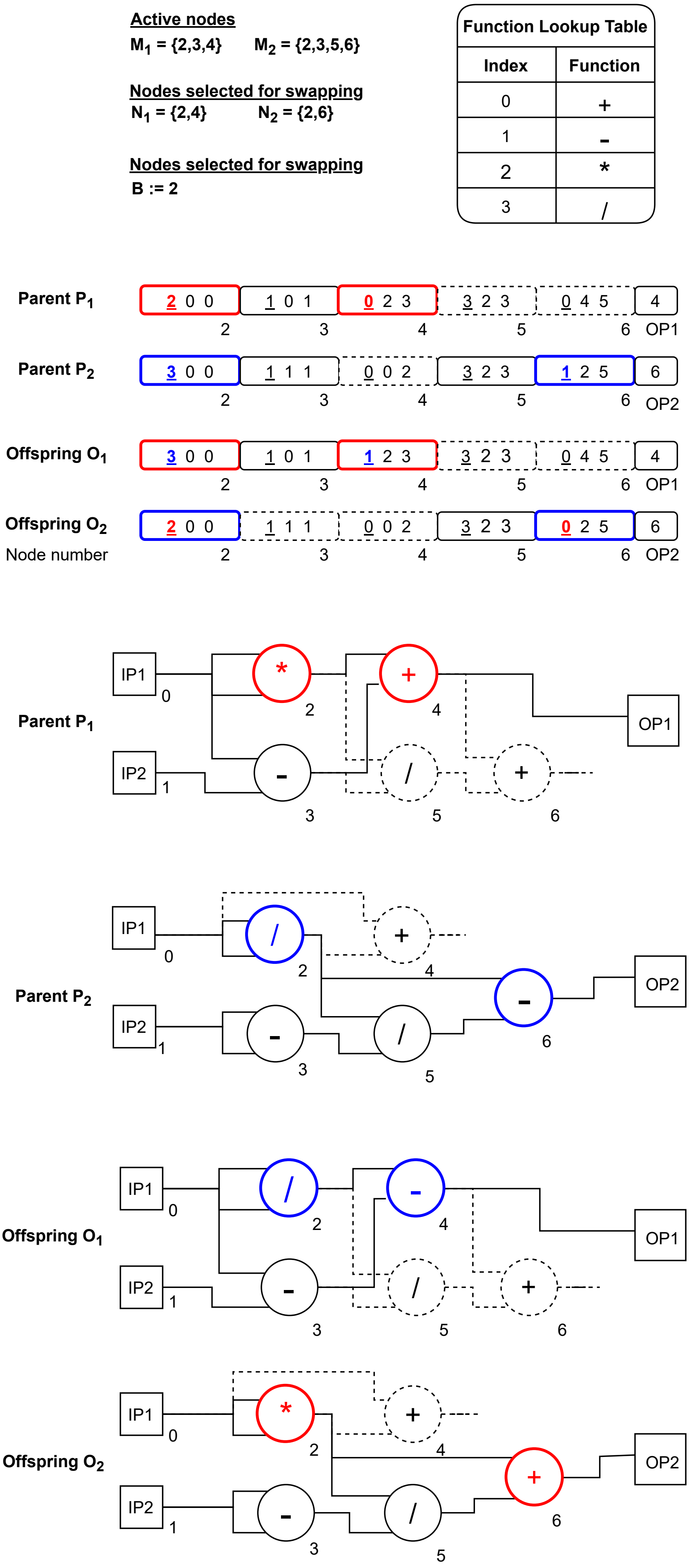
Definition 7.2 (Block crossover)**.** *The block crossover is a variation operator for CGP which exchanges blocks of function genes between two selected individuals, producing two offspring.*

The crossover procedure is described in more detail in Algorithms 7.6, 7.7 and 7.8. The function *DetermineSwapNodes* in Algorithm 7.6 describes the algorithmic implementation of the creation and handling of the active function blocks which are later used for the swapping of the respective function genes. The swapping itself is performed by the *SwapGenes* function as listed in Algorithm 7.7. Algorithm 7.8 describes the recombination process in which the functions *DetermineSwapNodes* and *SwapGenes* are called.

7.5 Evaluation of the proposed Methods

7.5.1 Experimental Setup

We performed experiments with symbolic regression, Boolean functions, and image operator design problems. To evaluate the search performance of the subgraph crossover, we measured the number of generations until the CGP algorithm terminates successfully (*generations-to-success*) and the best fitness value which was



Active nodes
M_1 = {2,3,4} **M_2 = {2,3,5,6}**

Nodes selected for swapping
N_1 = {2,4} **N_2 = {2,6}**

Nodes selected for swapping
B := 2

Function Lookup Table

Index	Function
0	+
1	-
2	*
3	/

Figure 7.2: The procedure of the proposed block crossover technique

Algorithm 7.6 Block crossover: Determinations of swap nodes

```
**Arguments**
$N_{\mathrm{A}}$: List of active nodes
$B$ : Predefined block size
**Return**
$N_{\mathrm{S}}$: List of active nodes which have been chosen for swapping

function DeterminSwapNodes($N_{\mathrm{A}}$,$B$)
    $j \leftarrow 0$                                  ▷ Initialize loop counter
    $n \leftarrow |N_{\mathrm{A}}|$                   ▷ Get the number of active nodes
    ▷ Initialize $N_P$, the list of possible nodes, with all active nodes from $N_{\mathrm{A}}$
    $N_P \leftarrow N_{\mathrm{A}}$
    $N_S \leftarrow$ EmptyList()                      ▷ Empty list for the nodes of the block
    while $j < B$ do                                  ▷ While $j$ is smaller than the block size
        $r \leftarrow$ RandomInteger(0,$n$)           ▷ Get a random index in the range 0 and $n$
        $n_r \leftarrow N_P[r]$                       ▷ Get the node number at radom index $r$
        $N_S$.append($n_r$)                           ▷ Append the node number to the block node list
        $N_P$.remove($n_r$)                           ▷ Decrease the selection of possible nodes
        $j \leftarrow j + 1$                          ▷ Increase loop counter
    end while
    return $N_S$                                      ▷ Return the block list
end function
```

Algorithm 7.7 Block crossover: The procedure for the swap of the function genes

Arguments

O_1: Genome of the first offspring

O_2: Genome of the second offspring

N_{S1}: Swap nodes of the first offspring

N_{S2}: Swap nodes of the second offspring

I: The number of input nodes

A: The maximum arity

Return

O_1, O_2: Genome of the first and second offspring

```
function SwapGenes($O_1$,$O_2$, $N_{S1}$, $N_{S2}$, $I$, $A$)
    $j \leftarrow 0$                                   ▷ Initialize loop counter

    while $j < B$ do                                   ▷ Iterate from 0 to $B - 1$
        $n_1 \leftarrow N_{S1}[j]$                     ▷ Get the node number at index $j$ from the block list $N_{S1}$
        $n_2 \leftarrow N_{S2}[j]$                     ▷ Get the node number at index $j$ from the block list $N_{S2}$

        $g_1 \leftarrow (n_1 - I) \cdot (1 + A)$       ▷ Calculate the function gene position of node $n_1$
        $g_2 \leftarrow (n_2 - I) \cdot (1 + A)$       ▷ Calculate the function gene position of node $n_2$

        $h \leftarrow O_1[g_1]$                        ▷ Save the function gene of the $P_1$ genome in $h$
        $O_1[g_1] \leftarrow O_2[g_2]$                 ▷ Transfer the function gene to be exchanged from $P_2$ to $O_1$
        $O_2[g_2] \leftarrow h$                        ▷ Transfer the function gene to be exchanged from $h$ to $O_2$

        $j \leftarrow j + 1$                           ▷ Increase loop counter
    end while
    return $O_1$, $O_2$                                ▷ Return the genomes of the offspring
end function
```

Algorithm 7.8 Block crossover: The recombination procedure

```
    Arguments
    $P_1$: Genome of the first parent
    $P_2$ : Genome of the second parent
    $B$ : The maximum block size
    $I$ : The number of input nodes
    $A$ : The maximum arity
    Return
    $O_1, O_2$: The genomes of the offspring

function BlockCrossover($P_1$,$P_2$, $B$, $I$, $A$ )
    $N_{A1} \leftarrow$ DetermineActiveNodes($P_1$)        ▷ Determine the active nodes of parent $P_1$
    $N_{A2} \leftarrow$ DetermineActiveNodes($P_2$)        ▷ Determine the active nodes of parent $P_2$

    $n_{a1} \leftarrow |N_{A1}|$        ▷ Determine the number of active nodes of parent $P_1$
    $n_{a2} \leftarrow |N_{A2}|$        ▷ Determine the number of active nodes of parent $P_2$

    if $n_{a1} = 0$ or $n_{a2} = 0$ then        ▷ Validate the number of active function nodes
        ▷ If at least one parent has no active function nodes, abort the crossover procedure
        return
    end if

    ▷ If at least one parent has less active nodes than the maximum block size
    if $n_{a1} < B$ or $n_{a2} < B$ then
        ▷ Determine the minimum of active function nodes between the two parents
        $B \leftarrow \min(n_{a1}, n_{a2})$
    end if

    ▷ Determine the blocks which will be swapped (Alg. 7.6)
    $N_{S1} \leftarrow$ DetermineSwapNodes($N_{A1}, B$)
    $N_{S2} \leftarrow$ DetermineSwapNodes($N_{A2}, B$)

    $O_1 \leftarrow P_1$        ▷ Initialize the first offspring with the genome of the first parent
    $O_2 \leftarrow P_2$        ▷ Initialize the second offspring with the genome of the second parent

    $O_1, O_2 \leftarrow$ SwapGenes($O_1, O_2, N_{S1}, N_{S2}, I, A$)        ▷ Perform the swaps (Alg. 7.7)
    return $O_1, O_2$        ▷ Return the offspring
end function
```

Table 7.1: Symbolic regression problems of the first experiment

Problem	**Objective Function**	**Vars**	**Training Set**
Koza-2	$x^5 - 2x^3 + x$	1	U[-1,1,20]
Koza-3	$x^6 - 2x^4 + x^2$	1	U[-1,1,20]
Nguyen-4	$x^6 + x^5 + x^4 + x^3 + x^2 + x$	1	U[-1,1,20]
Nguyen-5	$\sin(x^2)\cos(x) - 1$	1	U[-1,1,20]
Nguyen-6	$\sin(x) + \sin(x + x^2)$	1	U[-1,1,20]
Nguyen-7	$\ln(x + 1) + \ln(x^2 + 1)$	1	U[0,2,20]
Keijzer-6	$\sum_i^x 1/i$	1	E[1,50,1]
Pagie-1	$1/(1 + x^{-4}) + 1/(1 + y^{-4})$	2	E[-5,5,0.4]

found after a predefined number of generations (*best-fitness-of-run*). In addition to the mean values of the measurements, we calculated the standard deviation (SD) the median and the first and third quartile. To classify the significance of our results, we used the Mann-Whitney-U-Test. The mean values are denoted $a^\dagger$ if the p-value is less than the significance level 0.05 and $a^\ddagger$ if the p-value is less than the significance level 0.01 compared to the use of mutation as the sole genetic operator. We show the convergence behavior by plotting the average fitness function value against the number of generations. For this type of diagram, the fitness function value of the best solution was used.

Tournament selection was used to select new parent individuals. We performed preliminary experiments to determine the best configuration. A tournament size of four and seven individuals performed best for our experiments.

We performed 100 independent runs with different random seeds. Different rates of crossover were investigated, 0%, 20%, 50%, 70% and 90%. We also investigated different chromosome lengths, which are shown in the configuration of the experiments. The termination criteria are explained in the particular experiments. The levels back parameter l was set to ∞. For the block crossover we used a block size of 2 function nodes.

7.5.2 Symbolic Regression

For our first experiment, we chose eight symbolic regression problems from the work of Clegg et al. [13] and McDermott et al. [90] for better GP benchmarks. The functions of the problems are shown in Table 7.1. A training data set U$[a, b, c]$ refers to c uniform random samples drawn from a to b inclusive and E$[a, b, c]$ refers to a grid of points evenly spaced with an interval of c, from a to b inclusive. We replaced the investigation of the problem Koza-1 ("quartic") by the problems Keijzer-6, Nguyen-7, and Pagie-1, which have been proposed as alternatives by White et al. [160] to this overused problem and have different reputations.

For our first experiment, we chose eight symbolic regression problems from the work

Table 7.2: Configuration of the first experiment

Property	**Koza-2,3/Nguyen-4,5,6**	**Nguyen-7/Keijzer-6/Pagie-1**
Maximum node count	10	20/30/30
Maximum generation	-	1000
Number of inputs	2	2
Number of outputs	1	1
Population size	50	50
Function set	Koza	Koza/Keijzer/Koza
Mutation rate	0.2	0.05/0.04/0.04
Tournament selection size	4	4
Evaluation method	generations-to-success	best-fitness-of-run

of Clegg et al. [13] and McDermott et al. [90] for better GP benchmarks. The functions of the problems are shown in Table 7.1. A training data set U$[a, b, c]$ refers to c uniform random samples drawn from a to b inclusive and E$[a, b, c]$ refers to a grid of points evenly spaced with an interval of c, from a to b inclusive. We replaced the investigation of the problem Koza-1 ("quartic") by the problems Keijzer-6, Nguyen-7, and Pagie-1, which have been proposed as alternatives by White et al. [160] to this overused problem and have different reputations.
Tables 7.3 and 7.4 show the results of the first experiment, and it is visible that the use of the subgraph and block crossover reduces the number of generations until the termination criterion triggers and results in a better fitness value. The standard deviations are also clearly reduced by the use of higher crossover rates.

Figures 7.3 and 7.4 show the convergence behavior. It can also be clearly seen that the use of the subgraph and block crossover results in improved convergence curves for the presented number of generations.

7.5.3 Boolean Functions

To investigate the use of the subgraph crossover with one output in the Boolean domain, we chose the three Even-Parity problems with $n = 5$, 6, and 7 Boolean inputs. The goal was to find a program that produces the value of the Boolean even parity depending on the n independent inputs. The fitness was represented by the number of fitness cases for which the candidate solution failed to generate the correct value of the Even-Parity function.

Since former work by White et al. [160] outlined that this problem type was excessively used and investigated in the past, we also investigated multiple output problems as the digital adder, subtractor and multiplier. This sort of problem differs markedly from the parity problems, and the multiple output multiplier has been proposed as a suitable alternative. As a result, we receive a diverse set of problems in this domain.

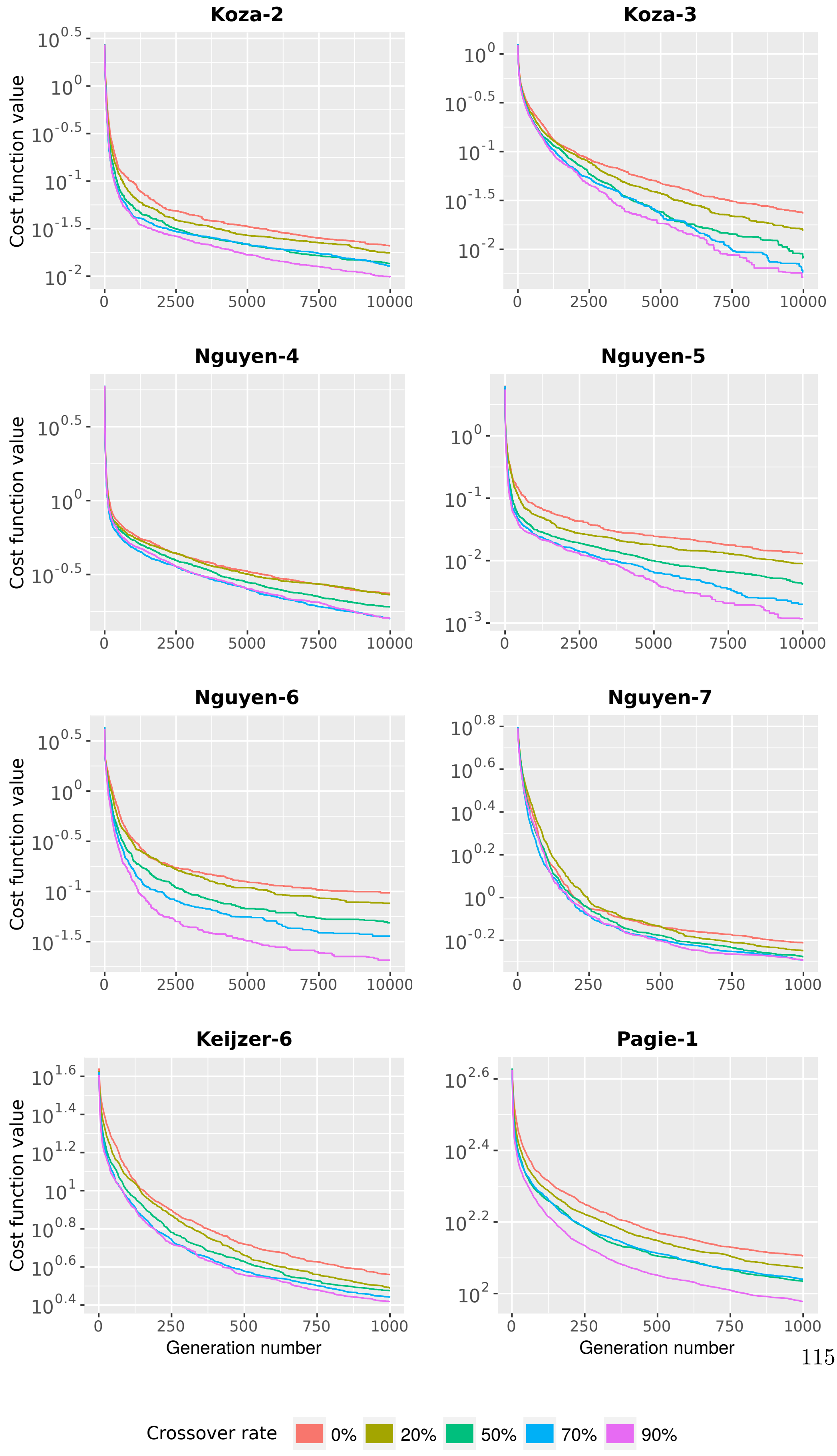


Figure 7.3: Convergence curves for all problems of the first experiment when subgraph crossover is used

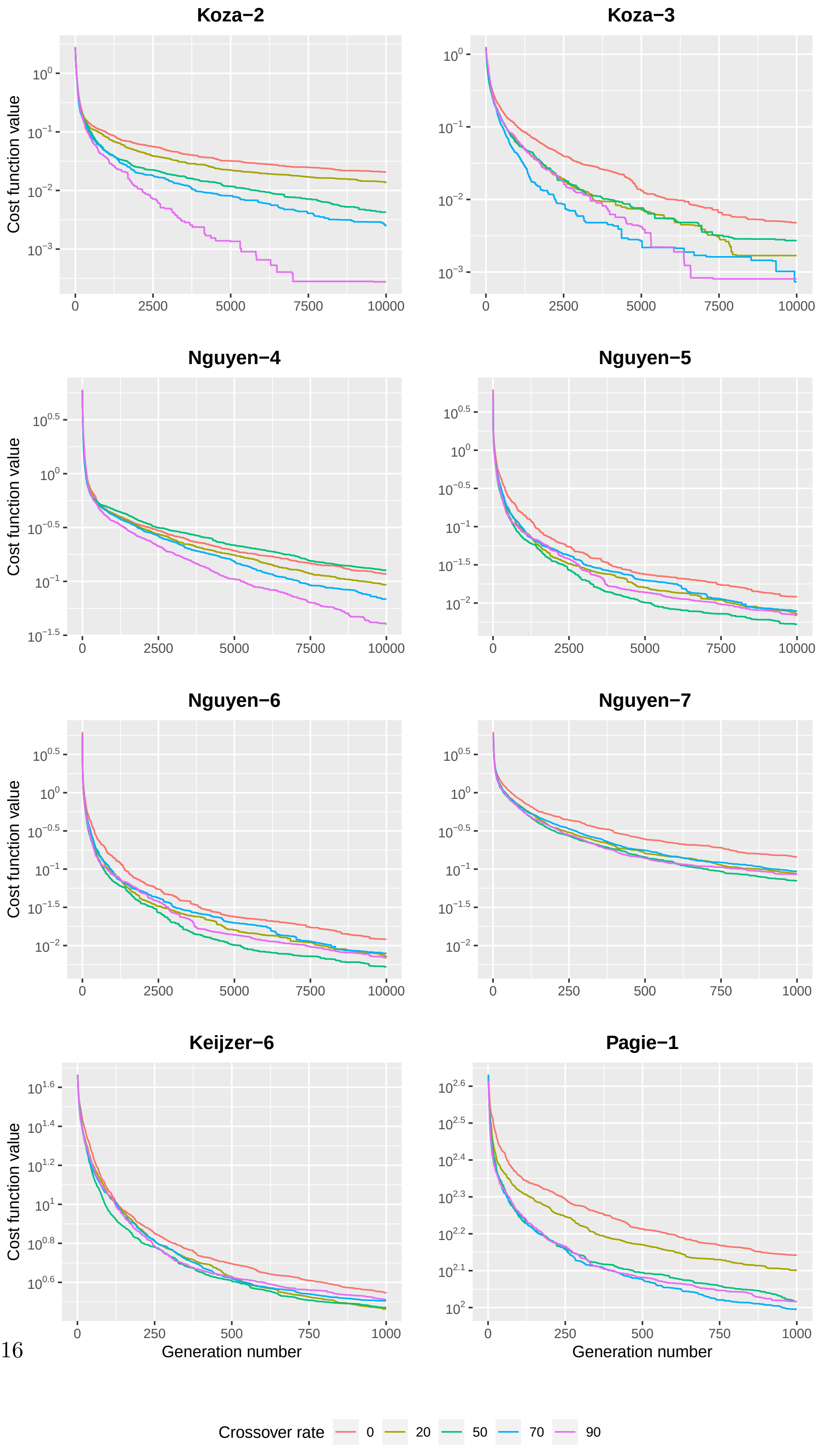


Figure 7.4: Convergence curves for all problems of the first experiment when block crossover is used

Table 7.3: Results for the symbolic regression problems Koza 2,3 and Nguyen 4,5,6 of the first experiments

Problem	Crossover	Crossover rate [%]	Mean Generations	SD	1Q	Median	3Q
Koza-2	Subgraph	0	23512	32265	4049	12627	30102
		20	16657	18745	4368	10243	21947
		50	10132‡	11636	2639	6140	14732
		70	8882‡	9570	1900	5734	112551
		90	**8642**‡	**10021**	**2419**	**5538**	**12194**
	Block	20	10436‡	10982	957	7805	15972
		50	5600‡	6262	794	3204	8284
		70	2535‡	4128	289	782	2729
		90	**1636**‡	**2166**	**288**	**677**	**2033**
Koza-3	Subgraph	0	9733	12645	1106	4662	12958
		20	5512‡	9649	700	1629	4281
		50	4044‡	5644	581	1709	5825
		70	4324‡	6836	697	1553	4967
		90	**2493**‡	**2456**	**546**	**1543**	**3996**
	Block	20	1282‡	1583	260	809	1816
		50	950‡	2331	144	347	752
		70	**820**‡	**2236**	**112**	**225**	**495**
		90	861‡	2120	185	363	657
Nguyen-4	Subgraph	0	2464455	3318452	270870	1286193	3777324
		20	1958761	2376462	293119	1002406	2536421
		50	434317‡	549211	43429	294556	580821
		70	182694‡	246221	19823	66307	260541
		90	**161754**‡	**190997**	**24570**	**95716**	**233140**
	Block	20	1705286	2359957	236279	779322	2049142
		50	286420‡	334450	58742	213758	360332
		70	195483‡	208052	27222	117247	271800
		90	**182651**‡	**220627**	**31223**	**93438**	**272722**
Nguyen-5	Subgraph	0	22703	63646	2281	6365	15032
		20	11883	28970	1366	4107	9395
		50	5453‡	6591	1282	2863	7556
		70	3772‡	4208	835	1905	5090
		90	**3262**‡	**4106**	**576**	**2007**	**4176**
	Block	20	9354	12387	1732	4866	12375
		50	5801‡	11099	911	2664	5920
		70	7747‡	14900	1531	3262	5669
		90	**6254**‡	**13885**	**1145**	**2571**	**7221**
Nguyen-6	Subgraph	0	98394	186370	1006	10462	94350
		20	47796†	108905	766	3517	43571
		50	24096‡	62941	591	2199	10848
		70	12480‡	37103	489	1109	5704
		90	**4552**‡	**12332**	**292**	**750**	**2040**
	Block	20	13872‡	30196	508	1748	10722
		50	7608‡	19366	679	1693	5786
		70	**6618**‡	**17202**	**679**	**1357**	**4209**
		90	7673‡	26876	650	1942	4367

Table 7.4: Results for the symbolic regression problems Nguyen-7, Keijzer-6, and Pagie-1 of the first experiment

Problem	**Crossover**	**Crossover rate [%]**	**Mean Best Fitness**	**SD**	**1Q**	**Median**	**3Q**
Nguyen-7	Subgraph	0	0.62	0.31	0.44	0.67	0.68
		20	0.57	0.26	0.39	0.65	0.68
		50	0.54†	0.28	0.36	0.56	0.67
		70	0.49†	**0.26**	0.29	0.53	0.67
		90	**0.49**‡	**0.30**	**0.28**	**0.45**	**0.67**
	Block	20	0.59	0.24	0.42	0.67	0.68
		50	0.51‡	0.23	0.34	0.55	0.67
		70	**0.50**‡	**0.23**	**0.35**	**0.53**	**0.68**
		90	0.52‡	0.35	0.28	0.55	0.68
Keijzer-6	Subgraph	0	4.06‡	1.84	2.67	3.94	5.38
		20	3.55†	1.87	2.64	3.02	3.94
		50	3.13†	1.30	2.57	2.89	3.81
		70	3.02‡	1.17	2.52	2.79	3.24
		90	**2.80**‡	**0.91**	**2.36**	**2.81**	**3.14**
	Block	20	3.27	3.64	1.80	2.30	3.36
		50	**2.87**‡	**1.29**	**2.06**	**2.75**	**3.26**
		70	2.97‡	1.44	2.21	2.93	3.35
		90	3.04‡	1.62	2.20	2.94	3.28
Pagie-1	Subgraph	0	138.77	51.86	91.32	153.05	171.94
		20	123.85†	52.05	77.46	118.51	160.51
		50	121.90†	47.13	80.13	124.67	158.51
		70	115.44‡	46.36	72.93	109.86	157.86
		90	**105.88**‡	**48.32**	**68.01**	**89.52**	**146.23**
	Block	20	115.99†	48.90	71.23	111.22	158.28
		50	110.25‡	47.65	66.17	102.30	152.58
		70	**97.52**‡	**44.68**	**62.10**	**77.46**	**128.10**
		90	98.03‡	42.46	62.52	77.46	140.21

Table 7.5: Configuration for the second experiment

Property	**Parity-5/6/7**	**Adder 1/2 Bit**	**Subtr. 2 Bit**	**Multipl. 2 Bit**
Maximum node count	100/200/200	30	30	30
Number of inputs	5/6/7	3/5	4	4
Number of outputs	1	2/3	3	4
Population size	50	50	50	50
Function set	AND, OR, NAND NOT, NOR	AND, OR, XOR NOR, AND*	AND, OR, XOR NOR, AND*	AND, OR, XOR NOR, AND*
Mutation rate	0.01	0.04	0.02	0.04
Tournament selection size	4/7/7	4	4	4

*AND with one inverted input

To evaluate the fitness of the individuals on the multiple output problems, we defined the fitness value of an individual as the number of different bits to the corresponding truth table. When this number became zero, the algorithm successfully terminated. The configuration for the experiment is shown in Table 7.5.

Tables 7.6 and 7.7 show the results of our second experiment, which demonstrates a reduced demand of generations until the algorithm terminates successfully on all tested problems, except the digital multiplier 2-Bit problem, by the use of the subgraph crossover. For the block crossover, it is visible that the use of this technique reduces the number of generations clearly for the parity problems. For the multiple-output problems, the use of the block crossover, reduced the number of generations significantly only on the 1-Bit Adder problem. Figures 7.5 and 7.6 provide boxplots of the results of the parity problems. Figures 7.7 and 7.8 provide boxplots for the results of the multiple output problems.

7.5.4 Image Operator Design

For our third experiment, we chose two image operator design problems from the field of evolvable hardware [137]. For these problems, we wanted to minimize the difference between a filtered image I_f and a reference image I_r, which were based on an input image I_i. The images I_i and I_r were of size $N \times M$ pixels. We used the pixel values of a 3×3 kernel matrix as the inputs of the cartesian program. Figure 3.13 in Chapter 3, Section 3.5 provides an example of an image operator evaluation procedure in CGP.
For the filtered image we took a size of $(N-2) \times (M-2)$ pixels. The reason for this is the convolution problem on the edges of the input image which is often treated by ignoring the edges. Let

$$\text{MDPP} := \frac{\sum_{i=1}^{N-2} \sum_{j=1}^{M-2} |I_f(i,j) - I_r(i,j)|}{(N-2)(M-2)}$$

be the mean difference per pixel between the reference image I_r and the filtered image I_f. If MDPP is zero, the images I_f and I_r are identical, except the pixels on

Table 7.6: Results for the single output Boolean function problems of the second experiment

Problem	**Crossover**	**Crossover rate** [%]	**Mean Generations**	**SD**	**1Q**	**Median**	**3Q**
Parity-5	Subgraph	0	140046	232897	18812	63434	153966
		20	144100	269526	19241	52155	151565
		50	76651‡	163377	8549	22494	68392
		70	**53552**‡	**73833**	**12624**	**27438**	**58253**
		90	59238‡	**115555**	10910	29951	59834
	Block	20	50461‡	66847	15320	24783	46199
		50	36909‡	38641	10215	21477	47070
		70	**31284**‡	**30653**	**10892**	**23175**	**40381**
		90	47927‡	62411	17296	28306	53631
Parity-6	Subgraph	0	231000	456531	28085	70396	205451
		20	172585	364813	22760	70833	224780
		50	136644	254251	24754	52091	115374
		70	176057	281030	23250	70369	204457
		90	**82437**‡	**120038**	**16192**	**37451**	**95735**
	Block	20	97750	96379	38929	67502	111229
		50	48000‡	41788	24187	34011	62665
		70	**40595**‡	**32630**	**16347**	**30984**	**53007**
		90	53465‡	47503	25219	40304	72420
Parity-7	Subgraph	0	546206	738980	95363	283023	602712
		20	362886‡	1183156	47202	118699	323969
		50	396758†	574841	58663	132519	435769
		70	226581‡	326875	57824	111351	284247
		90	**199142**‡	**242284**	**43430**	**101077**	**249247**
	Block	20	196584‡	197527	64510	120965	234348
		50	120744‡	107974	55620	88423	141738
		70	**115988**‡	**102361**	**53547**	**86416**	**142296**
		90	135588‡	117007	62643	104993	166933

Table 7.7: Results for the multiple-output Boolean function problems of the second experiment

Problem	Crossover	Crossover rate [%]	Mean Generations	SD	1Q	Median	3Q
Add.-1Bit	Subgraph	0	431	1042	62	194	435
		20	**259**†	**386**	**49**	**115**	**249**
		50	236	291	87	132	286
		70	272	300	89	165	346
		90	321	335	119	230	392
	Block	20	277	323	77	156	333
		50	**195**†	**215**	**76**	**141**	**250**
		70	478	656	103	231	529
		90	441	617	101	233	568
Add.-2Bit	Subgraph	0	24020	36796	3814	9540	26967
		20	19452	32228	3475	8658	20038
		50	12980	15538	4177	7678	13233
		70	16035	28576	3615	7316	16561
		90	**15102**‡	**28690**	**2303**	**5093**	**14763**
	Block	20	32653	33245	10135	21875	49101
		50	50691	50857	11846	26798	77675
		70	87045	119871	26242	51351	102212
		90	101499	101633	28668	66968	130013
Sub.-2Bit	Subgraph	0	35493	57016	5380	15025	40364
		20	24722	36585	6540.	13077	30838
		50	20014	27364	4661	8861	25205
		70	17930	20571	4917	11057	21731
		90	**15137**†	**20550**	**4185**	**9077**	**19768**
	Block	20	25839	38432	3336	12201	29760
		50	23332	27110	5563	14425	31679
		70	24945	28042	5122	17016	33450
		90	31954	25911	10591	26025	45643
Mul.-2Bit	Subgraph	0	9365	12104	1586	4830	11125
		20	6551	7629	2497	3965	8282
		50	7675	10117	2147	4251	9652
		70	8289	9723	2258	5191	11211
		90	9240	9255	2281	5097	14667
	Block	20	7232	6094	2937	5722	8858
		50	11251	10008	4374	7603	15581
		70	11793	9986	3942	9175	17181
		90	21632	21886	6949	14285	28999

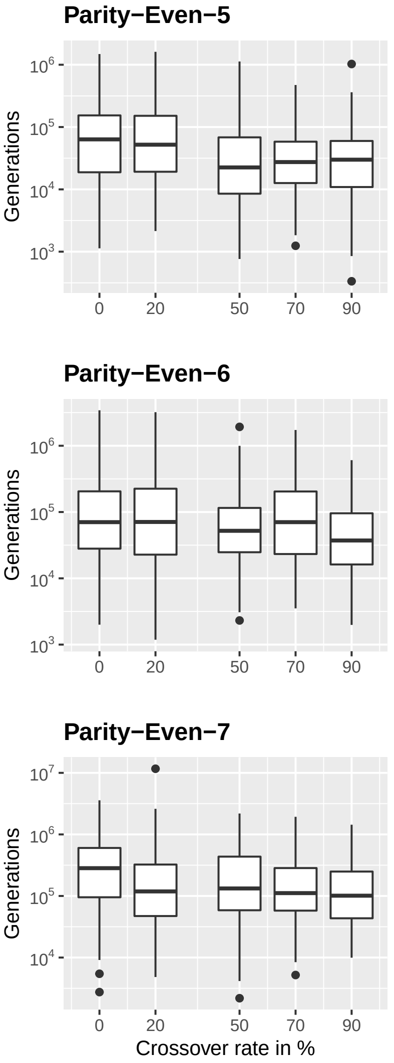


Figure 7.5: Boxplots for the parity problems evaluated with subgraph crossover

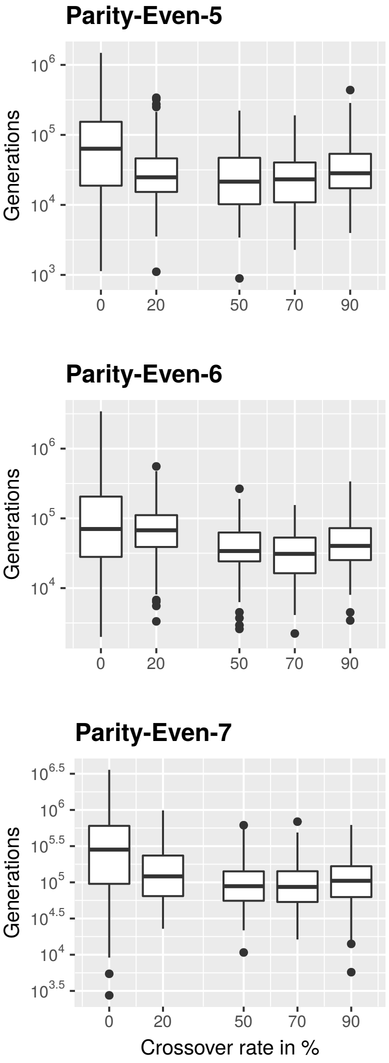


Figure 7.6: Boxplots for the parity problems evaluated with block crossover

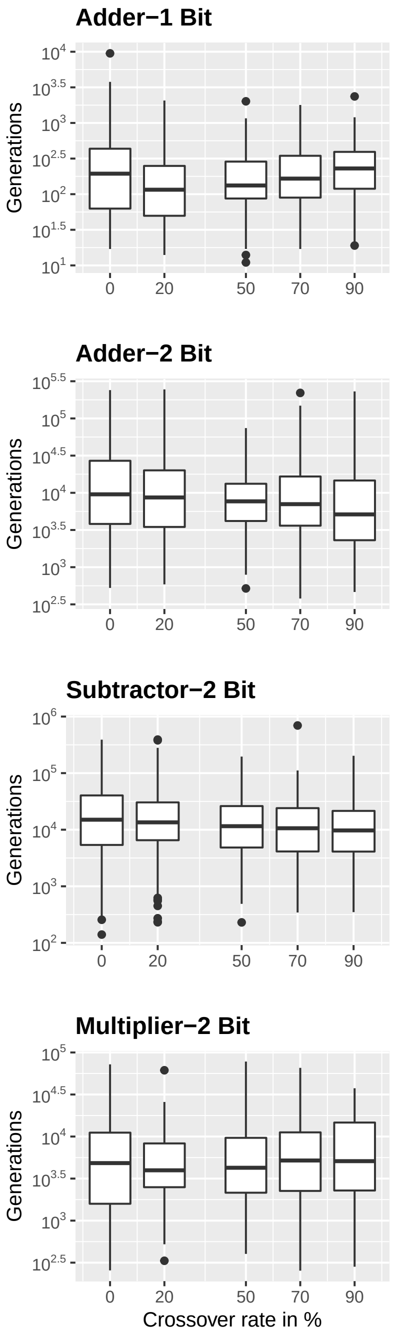


Figure 7.7: ¸Boxplots for the multiple output problems evaluated with subgraph crossover

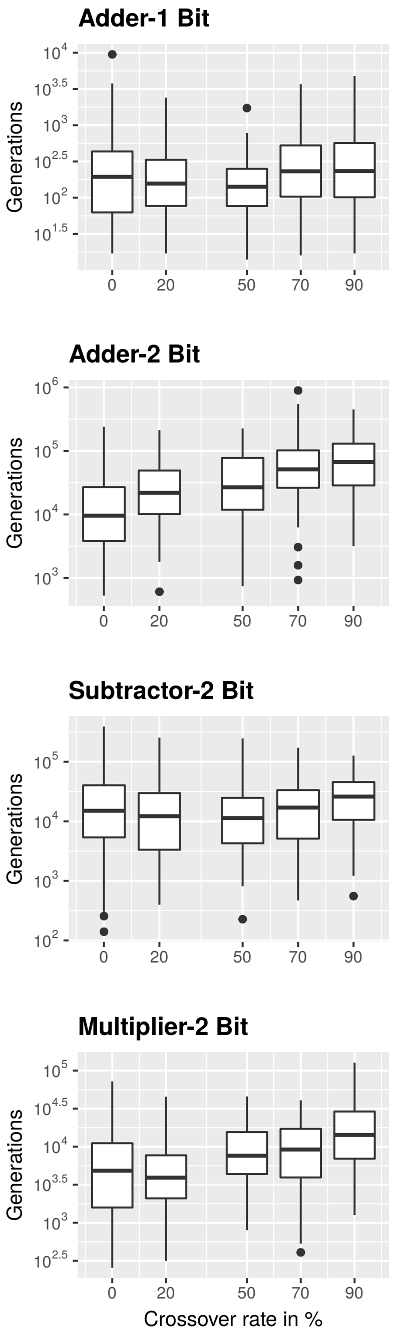


Figure 7.8: Boxplots for the multiple output problems evaluated with block crossover

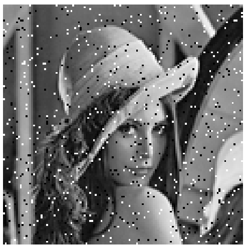


Figure 7.9: Salt & Pepper noise

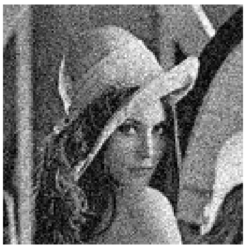


Figure 7.10: Gaussian noise

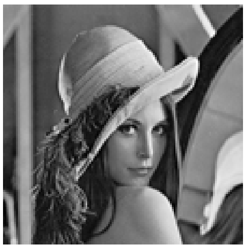


Figure 7.11: Original image

Table 7.8: Configuration of the third experiment

Property	**Configuration**
Maximum node count	20
Number of inputs	9
Number of outputs	1
Population size	100
Function set	OR, AND, XOR, ADD, ADD*, MAX, MIN, AVG
Mutation rate	0.05
Tournament selection size	4
Image size	64x64 pixels

*ADD with saturation

the edges. As a test image, we chose the famous Lena image. We added Gaussian Noise (Figure 7.10) and Salt & Pepper noise (Figure 7.9) to the Lena image and chose them as input images.
The goal of the resulting problem was to evolve a filter, which reduces the noise in reference to the Lena image without noise as shown in Figure 7.11. We defined the termination criterion with a MDPP of less than or equal 3. The configuration and function set for the experiment are shown in Table 7.8. The function set consisted of eight low-level image processing functions.

Table 7.9 shows the results of our third experiment, and it is visible that the use of the subgraph and block crossover leads to a reduced number of generations until the algorithm successfully terminated.
Figures 7.12 and 7.13 provide boxplots for the results of the image operator design problems.

7.5.5 Crossover Comparison

We compared the arithmetic crossover [13] to the subgraph crossover. We measured the generations until the termination criteria triggered (*generation-to-termination*) and the best fitness of run, similar to our previous experiments. The parameters for CGP were also similar. The crossover rate was set to 90%.
Table 7.10, 7.11, 7.12, 7.13 and 7.14 show the results of the crossover technique com-

Table 7.9: Results for the problems of the third experiment

Problem	**Crossover**	**Crossover rate** [%]	**Mean Generations**	**SD**	**1Q**	**Median**	**3Q**
Salt & Pepper noise	Subgraph	0	2187	2799	394	1335	2533
		20	1395‡	1829	258	627	1588
		50	1269‡	1660	249	781	1727
		70	1726	2192	405	1023	1896
		90	**1200**‡	**1418**	**245**	**786**	**1624**
	Block	20	1348	1443	338	827	1774
		50	**1079**‡	**1423**	**243**	**568**	**1406**
		70	1032‡	1044	273	630	1549
		90	1240†	1560	308	779	1510
Gaussian noise	Subgraph	0	3937	4373	1145	2459	4891
		20	2414†	2101	834	1846	3120
		50	3401	3366	808	2828	4869
		70	3203	2485	1518	2585	4174
		90	**2673**†	**2754**	**754**	**1884**	**3438**
	Block	20	3053	2884	731	2158	4291
		50	2954	2567	705	2297	4361
		70	**2501**‡	**2788**	**594**	**1766**	**3638**
		90	3241	3346	1091	2344	4108

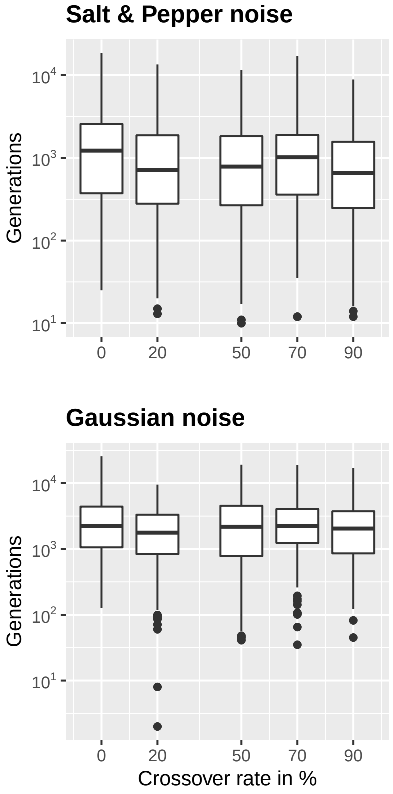


Figure 7.12: Boxplots for the image operator design problems evaluated with subgraph crossover

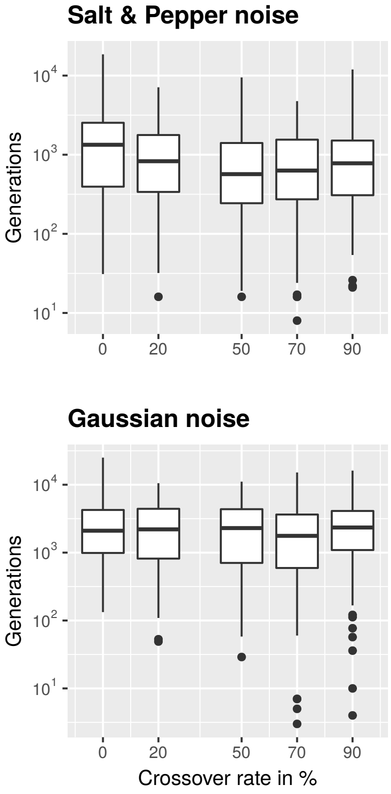


Figure 7.13: Boxplots for the image operator design problems evaluated with block crossover

Table 7.10: Results of the crossover comparison for the symbolic regression problems Koza 2,3 & Nguyen 4,5,6

Problem	Method	Mean Generations	SD	1Q	Median	3Q
Koza-2	Arithmetic	14688	23422	2746	8403	17551
	Subgraph	8642‡	10021	2419	5538	12194
	Block	**1636**‡	**2166**	**288**	**677**	**2033**
Koza-3	Arithmetic	2691	4305	376	964	3153
	Subgraph	2493‡	2456	546	1543	3996
	Block	**861**‡	**2120**	**185**	**363**	**657**
Nguyen-4	Arithmetic	8298115	10699730	739946	2856322	8715789
	Subgraph	**161754**‡	**190997**	**24570**	**95716**	**233140**
	Block	182651‡	220627	31223	93438	272722
Nguyen-5	Arithmetic	22829	55422	1976	7782	20733
	Subgraph	**3262**‡	**4106**	**576**	**2007**	**4176**
	Block	6254‡	13885	1145	2571	7221
Nguyen-6	Arithmetic	228610	547165	3171	44337	248089
	Subgraph	**4552**‡	**12332**	**292**	**750**	**2040**
	Block	7673‡	26876	650	1942	4367

Table 7.11: Results of the crossover comparison for the symbolic regression problems Nguyen-7, Keijzer-6, Pagie-1

Problem	Method	Mean Best Fitness	SD	1Q	Median	3Q
Nguyen-7	Arithmetic	0.64	**0.26**	0.55	0.67	0.68
	Subgraph	**0.49**†	**0.30**	**0.28**	**0.45**	**0.67**
	Block	0.52	0.35	0.28	0.55	0.68
Keijzer-6	Arithmetic	9.36	8.61	3.94	6.01	11.25
	Subgraph	**2.80**†	**0.91**	**2.36**	**2.81**	**3.14**
	Block	3.04†	1.62	2.20	2.94	3.28
Pagie-1	Arithmetic	128.95	48.77	80.12	137.34	160.38
	Subgraph	105.88‡	48.32	68.01	89.52	146.23
	Block	**98.03**‡	**42.46**	**62.52**	**77.46**	**140.21**

Table 7.12: Results of the crossover comparison for the single output Boolean function problems

Problem	**Method**	**Mean Best Generations**	**SD**	**1Q**	**Median**	**3Q**
Parity-5	Arithm.	898477	259872	1000000	1000000	1000000
	Subgraph	59238‡	115555	10910	29951	59834
	Block	**47927‡**	**62411**	**17296**	**28306**	**53631**
Parity-6	Arithm.	942262	206752	1000000	1000000	1000000
	Subgraph	92888‡	155648	16192	37451	95735
	Block	**53465‡**	**47503**	**25219**	**40304**	**72420**
Parity-7	Arithm.	1825487	504094	2000000	2000000	2000000
	Subgraph	199142‡	241070	43430	101077	249247
	Block	**135588‡**	**117007**	**62643**	**104993**	**166933**

Table 7.13: Results of the crossover comparison for the multiple output Boolean function problems

Problem	**Method**	**Mean Best Generations**	**SD**	**1Q**	**Median**	**3Q**
Add.-1Bit	Arithmetic	5036	15509	565	1026	3559
	Subgraph	**321‡**	**335**	**119**	**230**	**392**
	Block	441‡	617	101	233	568
Add.-2Bit	Arithmetic	99389	125828	28394	59583	111513
	Subgraph	**15102‡**	**28546**	**2303**	**5093**	**14763**
	Block	101499	101633	28668	66968	130013
Sub.-2Bit	Arithmetic	637929	4301481	135410	1000000	1000000
	Subgraph	**15137†**	**20550**	**4185**	**9077**	**19768**
	Block	31954‡	25911	10591	26025	45643
Mul.-2Bit	Arithmetic	43980	139033	10272	19017	29511
	Subgraph	**9240‡**	**9255**	**2281**	**5097**	**14667**
	Block	21632†	21886	6949	14285	28999

Table 7.14: Results of the crossover comparison for the image operator problems

Problem	**Method**	**Mean Best Generations**	**SD**	**1Q**	**Median**	**3Q**
Salt & Pepper noise	Arithmetic	5410	3463	2193	4797	9704
	Subgraph	**1175‡**	**1419**	**246**	**654**	**1571**
	Block	1240‡	1560	308	779	1510
Gaussian noise	Arithmetic	9762	**987**	10000	10000	10000
	Subgraph	**2383‡**	2086	**856**	**2045**	**3734**
	Block	3241‡	3346	1091	2344	4108

Table 7.15: Upper generation limits and success rates for the arithmetic crossover

Problem	**Upper generation count limit**	**Success rate**
Nguyen-4	$3 * 10^7$	0.83
Parity-5	10^6	0.16
Parity-6	10^6	0.08
Parity-7	$2 * 10^6$	0.10
Sub.-2Bit	10^6	0.45
Salt & Pepper noise	10^4	0.75
Gaussian noise	10^4	0.06

parison. Compared to the arithmetic crossover, the subgraph and block crossover perform significantly better on our tested problems. On some of the tested problems, the arithmetic crossover caused occasional runs, which took a huge number of generations. Consequently, we defined upper limits, which are shown in Table 7.15 with the corresponding success rates.

7.6 Fitness Improvement Analysis

To investigate possible conducive effects caused by the use of the crossover methods, we measured the number of fitness improvements on all benchmark problems of the evaluation part of this chapter. The total number of fitness improvements was increased in a measurement run if the offspring had a better fitness value than its parents. We performed 100 measurement runs for each problem and for each crossover method, which has been proposed in this chapter. Afterward, we calculated the median and the first and third quartiles. On the basis of these values, we illustrated the results of the measurements for various crossover rates with the help of boxplots. The boxplots for the symbolic regression problems are shown in Figures 7.14 and 7.15. Figures 7.16, 7.17, 7.18 and 7.19 illustrate the results for the Boolean function problems. Moreover, Figures 7.20 and 7.21 illustrate the result of the measurement for the image operator design problems. Even when it is visible that the use of both crossover methods leads to a higher median number of fitness improvements when compared to the measurements of the mutation-only CGP, it must be said that the outcome of the analysis cannot be used to make statements about conducive effects. In the first place, our findings are not coherent with the results of our search performance experiments, especially when it comes to the results of the respective crossover rates. The analysis also didn't include the magnitude of the improvements which should be utilized to allow more accurate findings.

7.7 A Comparative Study on Crossover

7.7.1 Experimental Setup

We have performed experiments on problems from the symbolic regression and Boolean function domains. To evaluate the search performance, we measured the

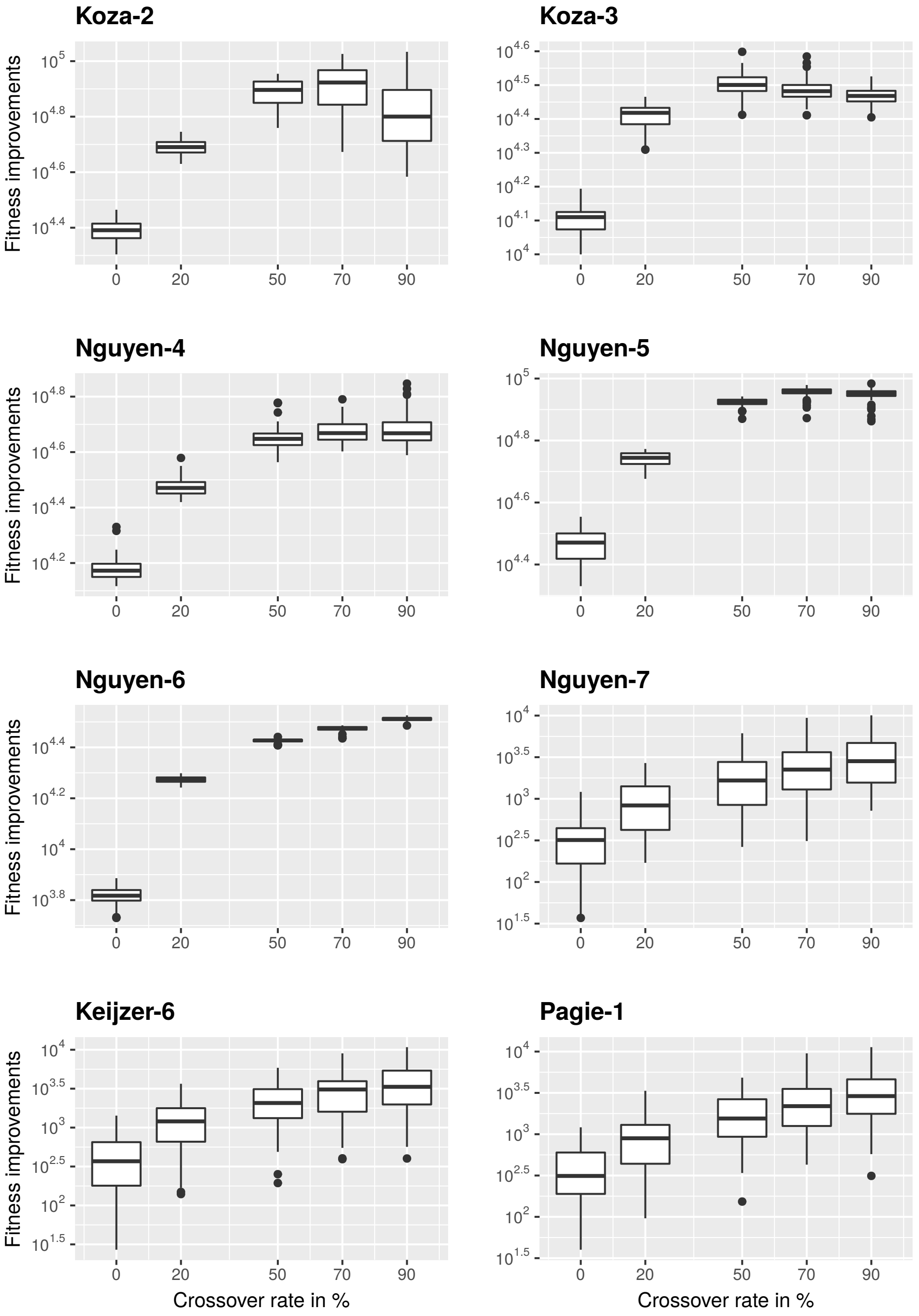


Figure 7.14: Boxplots for fitness improvements analysis on the symbolic regression problems when subgraph crossover is used.

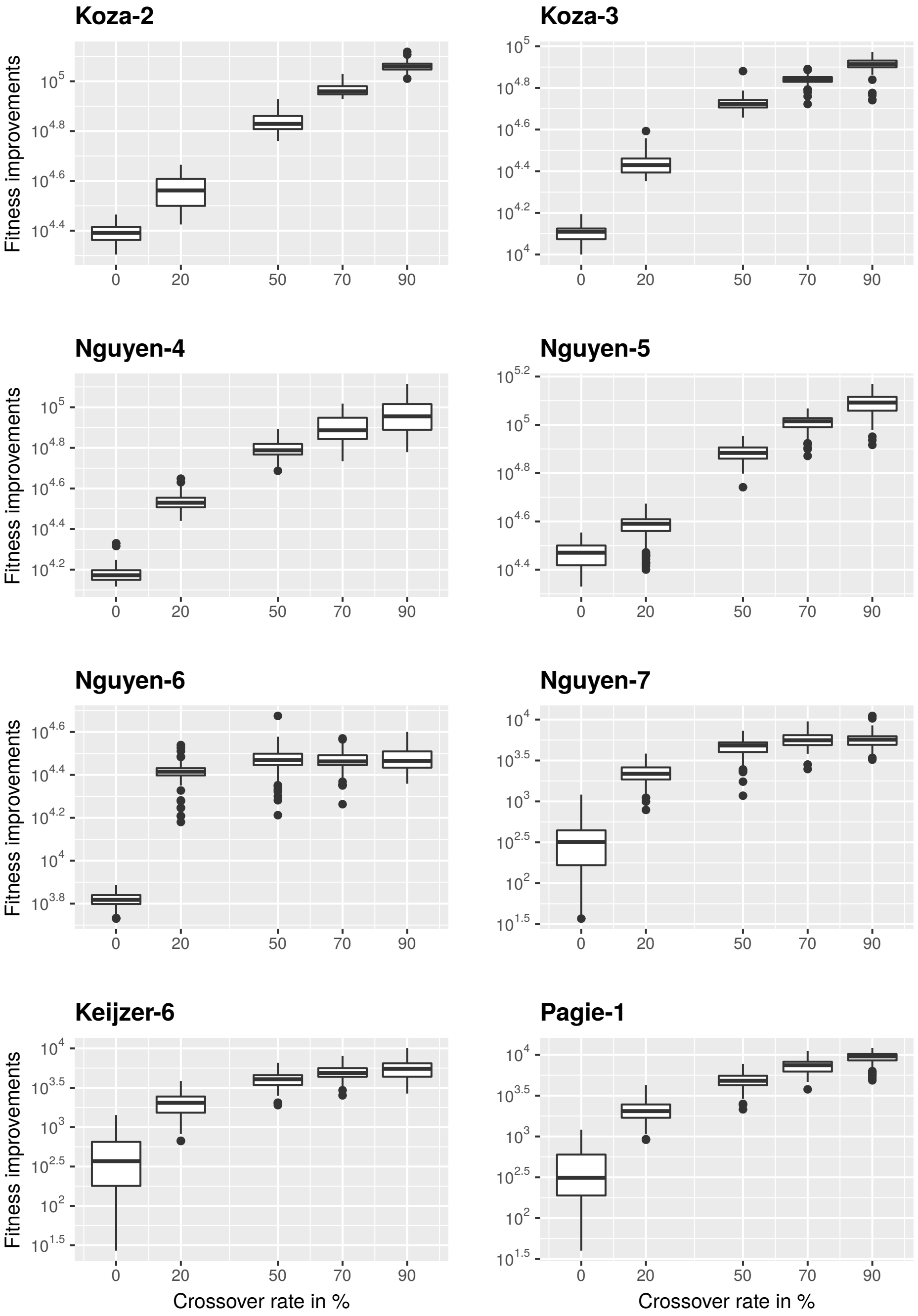
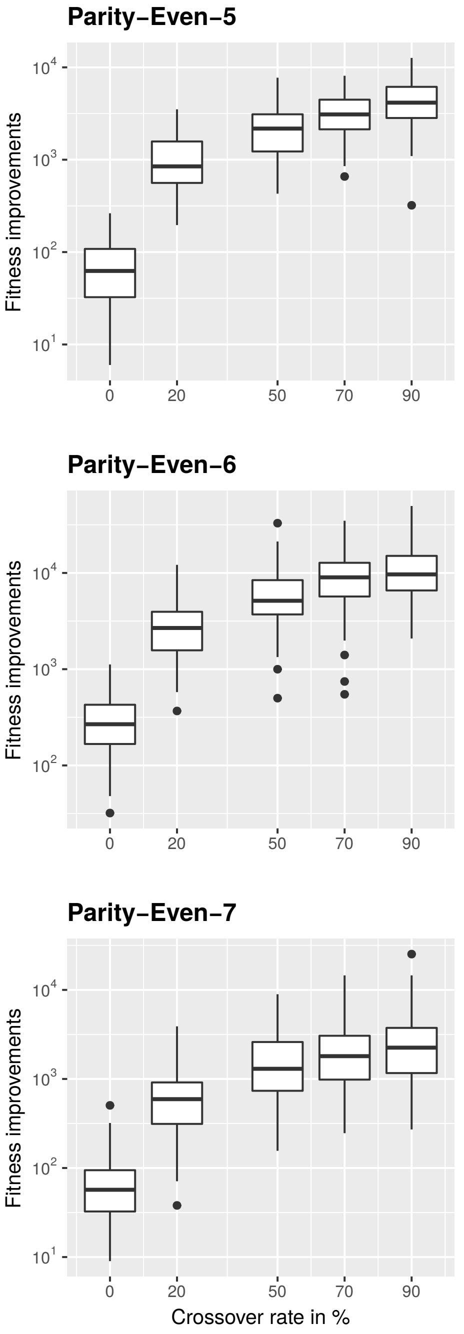


Figure 7.15: Boxplots for fitness improvements analysis on the symbolic regression problems when block crossover is used.



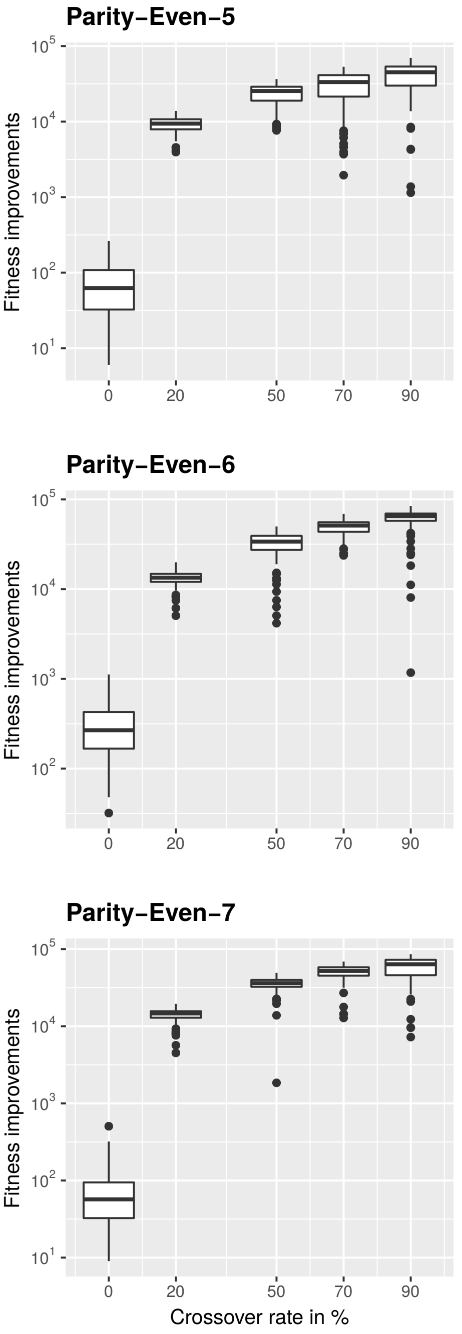


Figure 7.16: Boxplots for the fitness improvements analysis on the parity problems when subgraph crossover is used.

Figure 7.17: Boxplots for the fitness improvements analysis on the parity problems when block crossover is used.

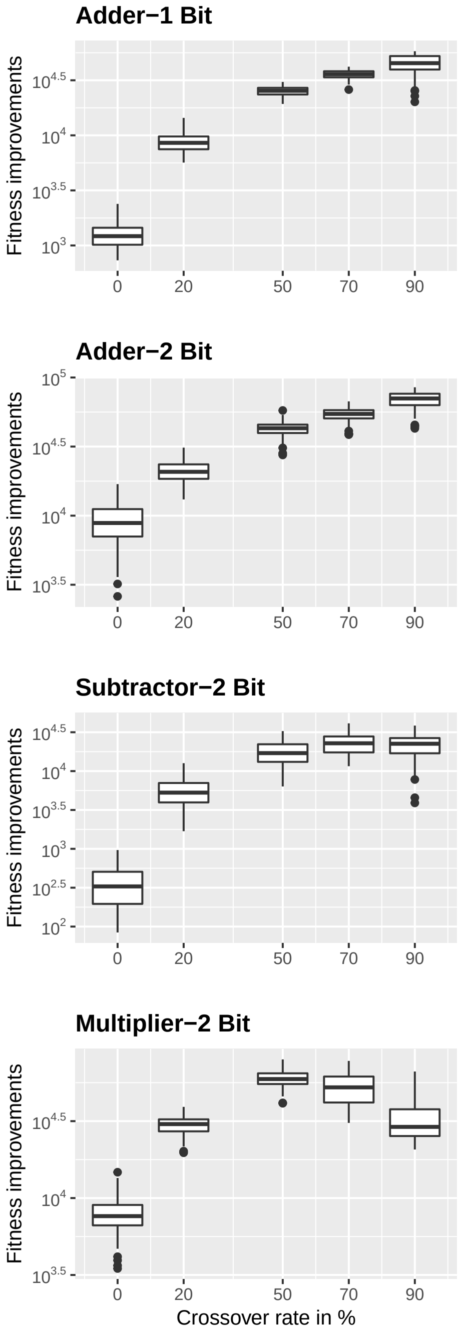


Figure 7.18: Boxplots for the fitness improvements analysis on the boolen multiple output problems when subgraph crossover is used.

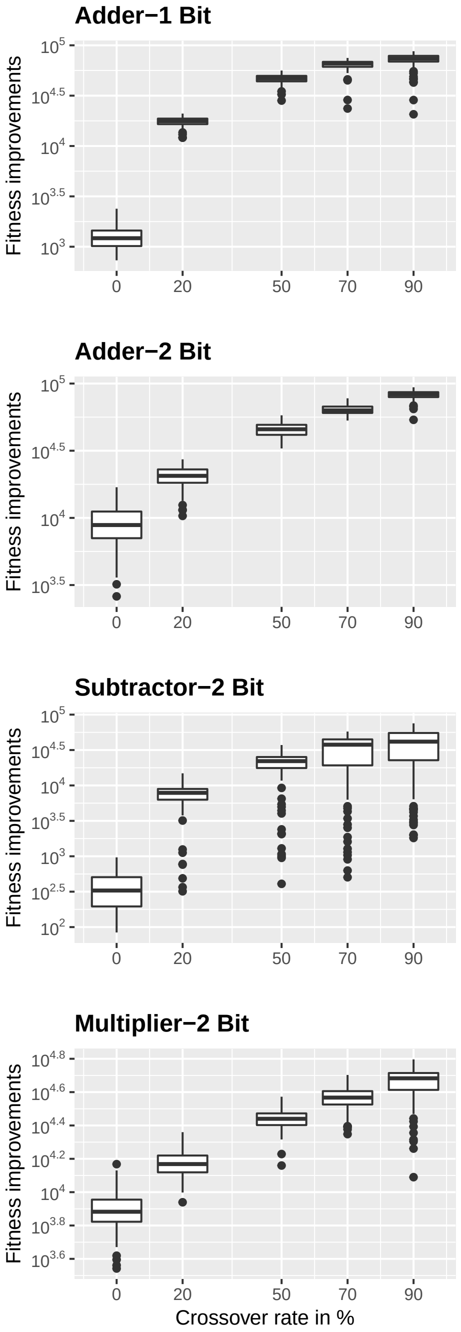


Figure 7.19: Boxplots for the fitness improvements analysis on the boolen multiple output problems when block crossover is used.

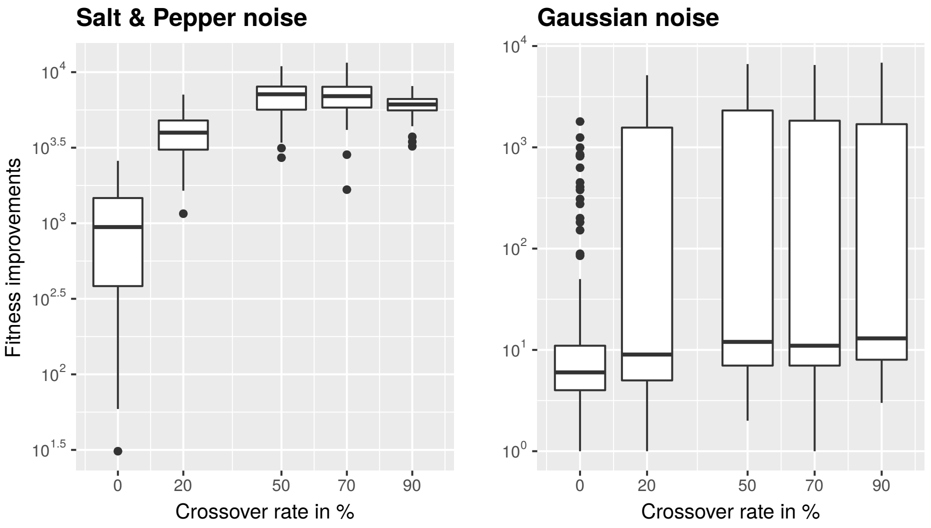


Figure 7.20: Boxplots for the fitness improvements analysis on the image operator design problems when subgraph crossover is used.

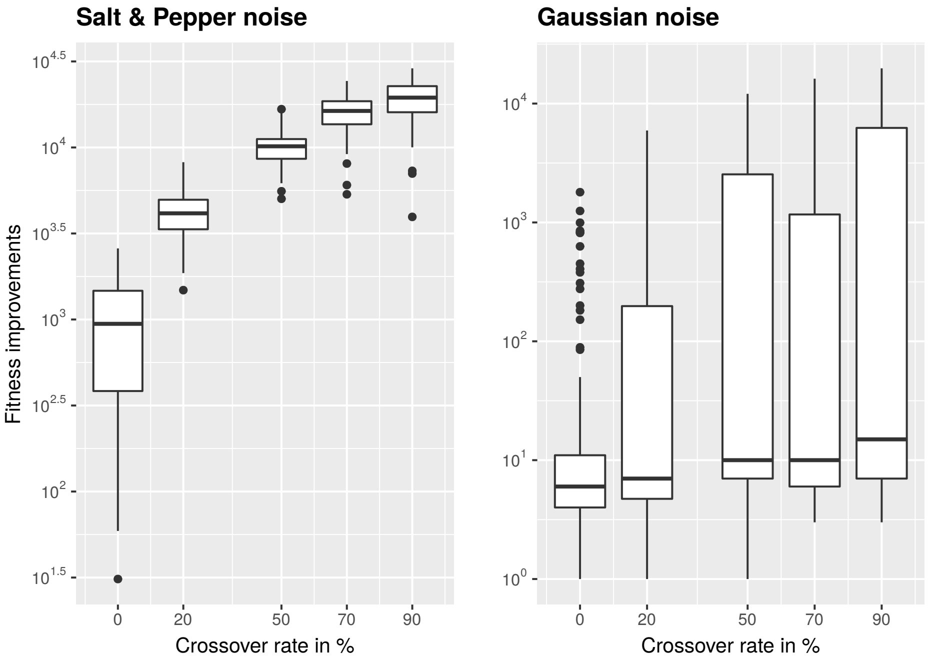


Figure 7.21: Boxplots for the fitness improvements analysis on the image operator design problems when block crossover is used.

best fitness value found after a predefined number of fitness function evaluations (*best-fitness-of-run*). For all problems, the fitness was to be minimized. Our comparison has focused on four crossover operators. standard one-point crossover, subgraph crossover, arithmetic crossover and block crossover.

The evolution used a generational model. The initial population was randomly generated. Parent genomes for the next generation are picked using two separate tournaments, which allow for the same individual to be picked multiple times. The parents and a crossover operator are used to create two offspring, which are then mutated. This process is repeated until a sufficient number of offspring has been created. The next generation consists of offspring and a certain percentage of the best individuals (elites) from the previous generation.

In addition, two more evolutionary setups were added for comparison. The none crossover setup uses the same evolutionary scheme, but the offspring it creates are identical clones of their parents, leaving mutation as the only active genetic operator. The $(1+\lambda)$ setup foregoes the above described setup and implements the traditional CGP algorithm.

We first performed two rounds of meta-evolutionary experiments in order to determine which evolutionary parameters were critical, so that the crossover operators can all use their optimal setting, and be compared in a fair way.

The two most important parameters were then subject to a parameter sweep, and for every crossover operator the best performing combination of parameters has been selected for comparison. To classify the significance of our results, we have used the Mann-Whitney U Test, to compare the standard $(1 + \lambda)$-CGP with all other crossover operators.

The implementation was done in Java, using the Java Evolutionary Computation Toolkit (ECJ) [84, 136]. All experiments were performed on a computing cluster with the following hardware configuration: 2 x Intel Xeon E5-2680v3 processor, 2.5 GHz, 12 cores; 128GB RAM, 5.3 GB cache per core, DDR4@2133 MHz; InfiniBand FDR56 network connection. The levels back parameter l was set to ∞.

7.7.2 Meta-optimization

For our experiments we used Meta-optimization, which can be described as an technique, which uses a certain optimization algorithm to tune the parameters of another optimization algorithm. Basically, meta-optimization is a simple way of finding performant parameters settings for a certain optimizing algorithm by using an overlaying optimizing algorithm which is called the meta-optimizer. Meta-optimization is considered to be first used in the late 1970s by Mercer and Sampson. For their work, Mercer and Sampson [94] used meta-optimization for finding the optimal parameter configuration of a genetic algorithm. Meta-optimization is also known as *meta-evolution* when evolutionary algorithms are used as the meta-optimizer. A so called meta-evolutionary algorithm (Meta-EA) is then an evolutionary algorithm which is used to optimize the parameters of a second underlying evolutionary algorithm. A meta-EA consists of two levels, a meta-level and an underlying base-level.

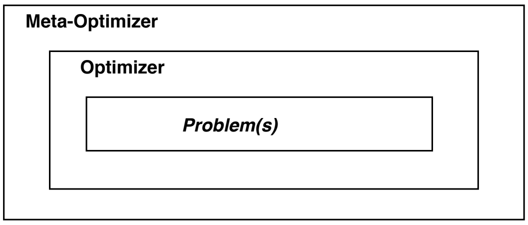


Figure 7.22: Meta-optimization

Table 7.16: Configuration of the first round of the meta-evolutionary GA.

Property	**Setting**	**Evolved parameter**	**Possible values**
Maximum generations	50	Mutation rate	0.01 – 0.20
Population size	10	Elitism rate	0, 0.05 – 0.50
Mutation probability	0.5	Population size	2 – 1024*
Mutation type	Random walk	Genotype length	2 – 1024*
Tournament size	2	Tournament size	2 – 1024*
Number of trials	5		

*\{2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128, 192, 256, 384, 512, 768, 1024\}.

At the base-level, the EA, which parameters are intended to optimize, is executed. The parameters which are optimized at the base-level are represented as individuals on the meta-level. The individuals of the meta-level are mainly represented as vectors. These vectors consist of values that fit a predefined parameter type and range. The values of the vectors are passed to the base-level EA. The quality or fitness of the Meta-EA's individuals can be measured by the fitness of the best base-level individual, which has been found after a predefined budget of fitness evaluations. A parameter tuning usually consists of two phases. The first phase is the evaluation of effective parameter settings. A meta-evolution system typically evolves a set of effective parameter settings. Afterward, these settings must be validated for the particular problem, and the best collection of parameters is selected afterward. Figure 7.22 illstrates the optimization structure with the respective meta and base levels.

For the meta-level, we used a basic canonical GA to tune five parameters we considered most important to the evolutionary process. Meta-evolution is very costly in terms of the computational effort necessary to find an optimal parameter setting. Furthermore, since GP benchmark problems can be very noisy in terms of finding the ideal solution, the evaluation of the evolved individuals is repeated multiple times, with fitness defined as the mean result. During the first round of meta-evolution, all problems used the same setting, and the evolved parameters have been limited to discrete values, as seen in Table 7.16. During the second round, the granularity and range were modified to better fit each individual problem. Because the $(1 + \lambda)$ scheme does not use tournament selection nor elitism, the two parameters have been ignored during its meta-evolution.

Results of the first round of meta-evolution have revealed that the tournament size parameter behaves wildly and does not converge to any specific value for any problem nor type of crossover. In some cases, it even significantly outgrew the population size. This caused the tournaments to include the entire population, resulting in a crossover of the best individual with itself, and wholly defeated the purpose of the crossover operator. To prevent this from happening, the tournament size has not been included in the second round of meta-evolution, and its value has been fixed to four.

Table 7.17 shows the results of the second round of meta-evolution which were used to set up the ensuing parameter sweep. Because the computational effort required to perform a parameter sweep grows exponentially with the number of parameters, only the two most important parameters, mutation rate and population size, were included in the sweep.

The ideal elitism rate was similar across all problems and types of crossover. For the sweep, it has been set to the overall average of 15%. Combined with the fixed tournament size of four, this means that during the sweep, there would be 52.2% chance that none of the individuals in a tourney would be elites from the previous generation. The ideal genotype length was highly variable and largely depended on the problem, rather than the type of crossover used. For the sweep, the genotype length was set up individually for each problem.

7.7.3 Boolean Functions

We have chosen to evolve both single and multiple output Boolean functions. 2-bit digital adder and multiplier were used as our multiple output problems. Former work by White et al. [160] proposed these, as suitable alternatives to the overused parity problems. Their fitness was defined as a hamming distance between the resulting truth table and the ideal solution. To increase the speed of the evaluation, we have used compressed truth tables.

For single output problems, we used 8-bit bent and 1-resilient Boolean functions. These functions find their use in cryptography, where they can provide an LFSR based key-stream generator of a stream cipher with resistance to linear and correlation attacks [11].

Bent Boolean functions possess the maximum possible degree of nonlinearity, defined as the Hamming distance between the truth table of a given function, and truth tables of all linear functions and their negations. For an 8-bit function, maximum degree of nonlinearity is 120 [116]. We defined their fitness, as the difference between its actual degree of nonlinearity and the optimal value.

1-resilient functions are highly nonlinear functions that are balanced and have correlation immunity of the first degree. Balancedness means that the function's truth table contains the same number of ones and zeros. Correlation immunity, means that if the truth table was split in half based on the value of a specific input, the two halves of the truth table would each remain balanced. To the best of our knowledge, the maximum possible nonlinearity of an 8-bit 1-resilient function is unknown, but it cannot be higher than 116 [133]. We defined the fitness, as the difference between the actual degree of nonlinearity and the optimal value, and if the evolved function

Table 7.17: Results of the second round of the meta-evolutionary GA. The table shows the best-performing combination of the four tuned parameters.

Problem	**Algorithm**	**Mutation**	**Elitism**	**Population**	**Genotype**
		rate	**rate**	**size**	**length**
Adder	$(1+\lambda)$	0.010	–	4	512
	None	0.01	0.080	4	384
	Block	0.010	0.100	4	1536
	Subgraph	0.010	0.060	6	768
	One-point	0.020	0.240	6	96
	Arithmetic	0.025	0.260	6	96
Multiplier	$(1+\lambda)$	0.050	–	3	24
	None	0.020	0.200	4	96
	Block	0.035	0.220	4	128
	Subgraph	0.040	0.040	4	64
	One-point	0.035	0.020	4	64
	Arithmetic	0.010	0.060	6	384
Bent	$(1+\lambda)$	0.140	–	24	128
	None	0.090	0.200	24	512
	Block	0.045	0.220	3	128
	Subgraph	0.040	0.200	12	256
	One-point	0.100	0.240	12	256
	Arithmetic	0.050	0.200	6	256
Resilient	$(1+\lambda)$	0.070	–	2	64
	None	0.070	0.200	32	2048
	Block	0.120	0.260	3	96
	Subgraph	0.090	0.260	3	96
	One-point	0.035	0.200	192	512
	Arithmetic	0.025	0.280	6	256
Koza-3	$(1+\lambda)$	0.080	–	24	64
	None	0.150	0.100	64	64
	Block	0.190	0.220	96	32
	Subgraph	0.070	0.200	14	16
	One-point	0.090	0.080	16	24
	Arithmetic	0.120	0.280	12	32
Nguyen-4	$(1+\lambda)$	0.070	–	24	192
	None	0.050	0.140	192	1024
	Block	0.110	0.080	6	96
	Subgraph	0.170	0.160	32	96
	One-point	0.180	0.160	6	128
	Arithmetic	0.050	0.100	16	128
Nguyen-7	$(1+\lambda)$	0.130	–	64	32
	None	0.110	0.180	12	96
	Block	0.100	0.100	6	48
	Subgraph	0.220	0.100	6	192
	One-point	0.090	0.280	64	256
	Arithmetic	0.160	0.120	4	48
Pagie-1	$(1+\lambda)$	0.050	–	2	384
	None	0.100	0.100	64	768
	Block	0.100	0.200	4	1536
	Subgraph	0.090	0.060	4	256
	One-point	0.050	0.080	8	1024
	Arithmetic	0.090	0.220	32	512

Table 7.18: Configuration of the Boolean function parameter sweep.

Property	Adder	Multiplier	Bent	Resileint
Input bits	5	4	8	8
Output bits	3	4	1	1
Genotype length	512	96	256	192
Mutation rate	0.002 – 0.02	0.005 – 0.05	0.01 – 0.1	0.01 – 0.1
Population size	2 – 48⋆	2 – 48⋆	2 – 48⋆	2 – 48⋆
Fitness evaluations	10000	5000	2000	5000
Tournament size	2	2	2	2
Percentage of elites	0.15	0.15	0.15	0.15

⋆{2, 3, 4, 6, 8, 12, 16, 24, 32, 48}.

was not resilient, its fitness was further penalized by 58, half the known limit.

Table 7.18 shows the setting used for the parameter sweep of Boolean functions. Each setting was run one hundred times, for every problem and type of crossover. All problems used the following function set AND, OR, XOR, AND with one input inverted. Because the best performing population size was usually very small, we have reduced the tournament size to two, to avoid repeating the issue from the first round of meta-evolution. For problems where the optimized setting could routinely find the ideal solution, we have also reduced the number of fitness function evaluations to get more telling results.

Table 7.19 shows the results of the parameter sweep. For each problem and crossover operator, we have selected combination of mutation rate and population size, which provided the best mean fitness over the hundred runs. Those that performed significantly different from $(1+\lambda)$, have their mean values marked. The table also shows the standard deviation (SD) and three quantiles.

Figure 7.23 provides visual comparison using box plots. The Arithmetic crossover, originally intended for use in symbolic regression, performs the worst when used for Boolean function design. For adder and multiplier problems, the $(1+\lambda)$ strategy has significantly surpassed all other approaches. However, for the bent function, there was no statistically significant difference, and for the 1-resilient function, the $(1+\lambda)$ has performed significantly worse than the other options. Here, even with an optimal setting, some runs failed to produce a resilient function, resulting in significant deterioration of the average fitness.

7.7.4 Symbolic Regression

For symbolic regression, we have chosen four problems from the work of Clegg et al. [13] and McDermott et al. [90] for better GP benchmarks, and the Pagie-1 one problem proposed by White et al. [160] as an alternative to the heavily overused Koza-1 (“quartic") problem. The analytic functions of the problems are shown in Table 7.20. The training data set U$[a, b, c]$ refers to c uniform random samples drawn from a to b inclusive and E$[a, b, c]$ refers to a grid of points evenly spaced with an interval of c, from a to b inclusive.

The fitness of the individuals was represented by a cost function value, defined as the

Table 7.19: Results of the parameter sweep for Boolean functions.

Problem	Crossover	Mutation	Pop.	Mean	SD	Q1	Median	Q3
	type	rate	size	fitness				
Adder	$(1+\lambda)$	0.010	2	**4.26**	3.39	1	4	6
	None	0.008	3	6.61‡	3.46	4	6	9
	Block	0.010	3	6.88‡	3.24	5	7	8
	Subgraph	0.010	3	6.60‡	3.91	4	6	9
	One-point	0.014	2	6.99‡	3.56	4.75	6.5	8.25
	Arithmetic	0.010	3	6.96‡	3.10	5	7	9
Multiplier	$(1+\lambda)$	0.035	2	**1.13**	1.00	0	1	2
	None	0.020	4	2.09‡	1.48	1	2	3
	Block	0.035	3	2.14‡	1.54	1	2	3
	Subgraph	0.015	3	1.85‡	1.42	1	2	3
	One-point	0.020	4	2.03‡	1.50	1	2	3
	Arithmetic	0.025	2	2.23‡	1.52	1	2	3
Bent	$(1+\lambda)$	0.050	2	**2.92**	3.86	0	0	8
	None	0.060	24	4.10	4.37	0	4	8
	Block	0.040	8	3.89	4.00	0	4	8
	Subgraph	0.050	32	4.28	4.19	0	4	8
	One-point	0.050	16	4.04	3.92	0	4	8
	Arithmetic	0.050	3	4.88	4.09	0	8	8
Resilient	$(1+\lambda)$	0.070	2	16.89	19.66	4	4	20
	None	0.070	4	**5.84**‡	5.06	4	4	4
	Block	0.080	6	6.64‡	5.69	4	4	4
	Subgraph	0.040	6	6.24‡	5.46	4	4	4
	One-point	0.090	4	6.12‡	5.28	4	4	4
	Arithmetic	0.040	3	8.48†	6.94	4	4	14

† p-value is less than 0.05. ‡ p-value is less than 0.01.

Table 7.20: Symbolic regression problems used in the experiment.

Problem	Objective function	Vars	Training set
Koza-3	$x^6 - 2x^4 + x^2$	1	U[-1, 1, 20]
Nguyen-4	$x^6 + x^5 + x^4 + x^3 + x^2 + x$	1	U[-1, 1, 20]
Nguyen-7	$ln(x+1) + ln(x^2+1)$	1	U[0, 2, 20]
Pagie-1	$1/(1+x^{-4}) + 1/(1+y^{-4})$	2	E[-5, 5, 0.4]

Table 7.21: Configuration of the symbolic regression parameter sweep.

Property	Koza-3	Nguyen-4	Nguyen-7	Pagie-1
Genotype length	48	128	128	512
Mutation rate	0.02 – 0.2	0.02 – 0.2	0.02 – 0.2	0.02 – 0.2
Population size	4 – 96*	4 – 96*	4 – 96*	4 – 96*
Fitness evaluations	10000	10000	10000	10000
Tournament size	4	4	4	4
Percentage of elites	0.15	0.15	0.15	0.15

* {4, 6, 8, 12, 16, 24, 32, 48, 64, 96}.

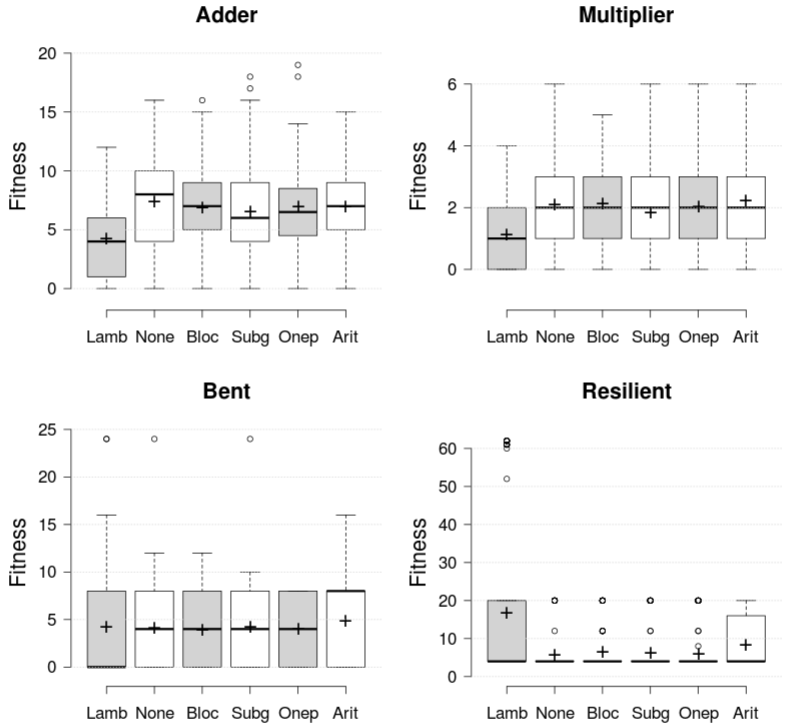


Figure 7.23: Comparison of crossover operators for Boolean functions.

sum of the absolute differences between the correct function values and the values of an evaluated individual. The configuration of the experiment is shown in Table 7.21. All problems used the following set of mathematical functions $\{+, -, *, /, \sin, \cos, \ln(|n|), e^n\}$.

Table 7.22 shows the parameter sweep results. Same as before, the primary selected criterion was the best average fitness over one hundred runs. Crossover operators that performed significantly different from $(1 + \lambda)$, have their mean values marked. As can be seen in Figure 7.24, the arithmetic crossover performs very well, when used for symbolic regression, as originally designed.

7.8 Discussion

The primary concern of our experiments was to find significant contributions of the proposed subgraph and block crossover techniques to the search performance of CGP in different problem domains. One point which should be discussed is the observation of the crossover rate in Section 7.5. On the nearly all of the tested problems, a medium and a high crossover rate led to the best results. However, on some problems, a low crossover rate led to the best results or no improvement of the search performance was achieved.

For this behavior, we have no general and provable explanation yet. A preliminary

Table 7.22: Results of the parameter sweep for symbolic regression.

Problem	Crossover type	Mutation rate	Pop. size	Mean fitness	SD	Q1	Median	Q3
Koza-3	$(1+\lambda)$	0.16	24	0.07	0.08	0.01	0.05	0.08
	None	0.12	16	0.06	0.08	0.01	0.04	0.08
	Block	0.06	12	0.06	0.08	0.02	0.05	0.08
	Subgraph	0.16	64	0.07	0.08	0.02	0.05	0.08
	One-point	0.14	96	0.06	0.06	0.02	0.03	0.09
	Arithmetic	0.12	12	**0.04**	0.04	0.01	0.03	0.08
Nguyen-4	$(1+\lambda)$	0.12	6	**0.31**	0.27	0.16	0.25	0.38
	None	0.10	8	0.33	0.23	0.17	0.29	0.41
	Block	0.08	16	0.35	0.28	0.18	0.29	0.41
	Subgraph	0.10	6	$0.37^{\dagger}$	0.29	0.19	0.34	0.49
	One-point	0.10	12	0.33	0.24	0.16	0.29	0.42
	Arithmetic	0.08	8	0.32	0.23	0.16	0.26	0.43
Nguyen-7	$(1+\lambda)$	0.18	64	**0.67**	0.43	0.43	0.59	0.77
	None	0.10	6	0.69	0.37	0.45	0.61	0.81
	Block	0.12	24	$0.76^{\dagger}$	0.34	0.55	0.72	0.92
	Subgraph	0.12	32	$0.77^{\dagger}$	0.45	0.51	0.72	0.96
	One-point	0.16	16	0.71	0.3741	0.44	0.70	0.84
	Arithmetic	0.14	6	$0.81^{\dagger}$	0.50	0.55	0.70	0.83
Pagie-1	$(1+\lambda)$	0.08	8	130.88	48.22	93.74	122.79	160.89
	None	0.06	96	134.51	46.70	96.31	140.63	170.56
	Block	0.04	96	126.11	45.79	87.47	122.37	161.16
	Subgraph	0.08	64	$150.48^{\ddagger}$	46.92	115.01	161.96	181.65
	One-point	0.06	8	130.61	48.96	96.44	122.49	169.57
	Arithmetic	0.04	8	**120.15**	45.71	84.60	114.43	152.56

† p-value is less than 0.05. ‡ p-value is less than 0.01.

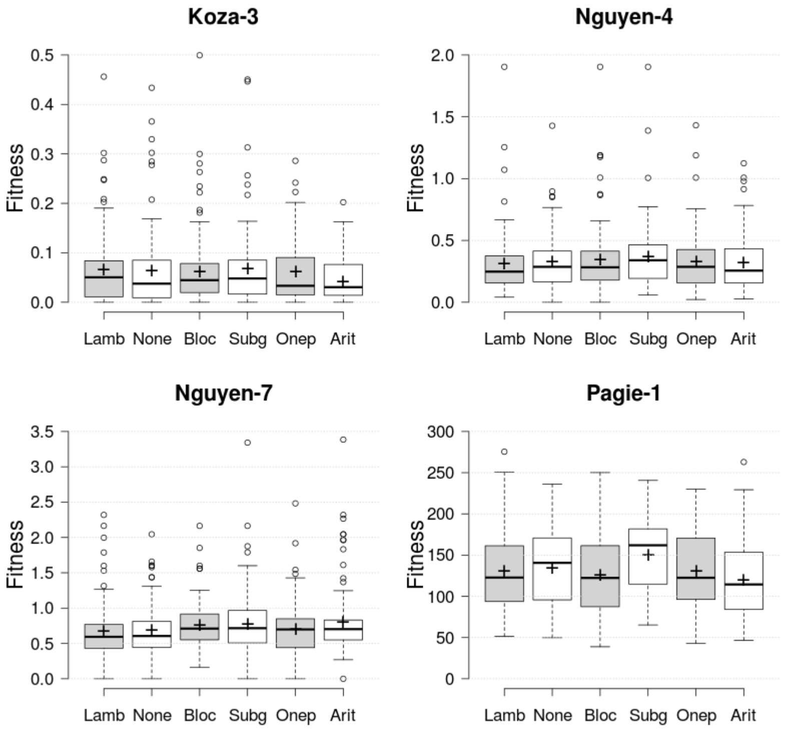


Figure 7.24: Comparison of crossover operators for symbolic regression.

and hypothetical assumption could be that certain combinatorial structures of the respective problems are susceptible to genetic variation. For instance, on the multi-output problems, multiple outputs lead to various semantics. On some of these problems variation steps which are caused by the subgraph and block crossover could be even more disruptive compared to single output structures. This assumption is backed by the observation that the block crossover did not contribute to the search performance on the majority of the tested multi-output problems. However, for more significant statements, a more detailed investigation of the subgraph crossover is needed. It should be based on the automated tuning of the crossover rate on a diverse set of problems.

Another point which should be discussed is the population size. For our experiments, we used sizes that are oriented with former work on CGP [13] and sizes which are empirically determined. However, it should be investigated which population sizes perform most efficiently in different problem domains when the crossover is used. Former work by Miller [97] outlined that very small population sizes can be very efficient for Boolean function problems. Based on Miller's experiments, we plan to investigate the use of the subgraph crossover with a $(\mu + \lambda)$ evolutionary algorithm and small population sizes. Concerning the selection pressure, we would like to point out that the size of the tournament in our experiment was based on empirical observations and should be taken with caution.

Our results raise the question in which way the subgraph crossover improves the search performance of CGP under certain conditions. At this time, we can only provide a preliminary and hypothetical suggestion. We assume that the use of the subgraph crossover leads to more exploration of high fitness regions in the search space. When parent graphs of high fitness are recombined, the number of diverse children with high fitness may be increased.Related to our comparison to the arithmetic crossover, we would like to point out that the parameters for the comparison were determined empirically. For a more significant comparison, an automated tuning of the CGP parameters for both techniques is necessary and should be included in future work.

In our meta-optimization experiments of the presented comparative study in Section 7.7, we dealt with significant problems to make a fair comparison. We were able to determine optimal parameter settings for the $(1 + \lambda)$-CGP as the tuning consists of only three parameters: population size, mutation rate, and genotypic length. However, determining optimal parameter settings for the canonical crossover-based algorithms is more complex. There are three additional parameters to contend with: crossover rate, elitism rate, and tournament size, which makes obtaining an optimal parameter setting for the respective problems significantly more difficult.

Furthermore, former studies on the traditional $(1 + \lambda)$-CGP algorithm have shown that large genotypes are very effective for the performance of CGP for certain problems. Consequently, we have to deal with a big parameter space in CGP to determine the optimal parameters and to make a fair comparison.

Another point which should be discussed is the observation that each type of crossover works best with different settings. Our findings indicate that there exists no general parameterization pattern for CGP when the crossover is in use. We think it should be investigated if there are similar behaviors like exploration abilities which could be obtained by fitness and search space analyses.

The results of our study show that the parameter settings vary for different problems in the respective problem domain and indicate that there is no general pattern to parametrize the $(1+\lambda)$-CGP in a well-performing way. These findings also open up a new question, which conditions or types of problems have the need for bigger or smaller population sizes. A preliminary assumption could be that the fitness landscape of certain problems requires more exploration abilities in order to overcome local optima.

Our results indicate that bigger populations perform well in the symbolic regression domain. This finding is consistent with the results of Chapter 5 which also indicates that bigger populations perform best in the symbolic regression domain.

Since our experiments validate the results of Chapter 5, this behavior should be investigated through more detailed experiments. Furthermore, we think that these findings offer a good opportunity to get more understanding of how CGP works in detail and can significantly contribute to the overall knowledge of the fitness landscape analysis in CGP.

Specifically, the results in Chapter 5 show that a mutation-only $(\mu+\lambda)$ evolutionary algorithm with a bigger population size can be very effective. Therefore, we think it should be investigated whether the subgraph and the block crossover can be used with a $(\mu+\lambda)$ evolutionary algorithm, like a part of our attempts to proceed toward more precise comparative studies in CGP.

7.9 Conclusion

In this chapter, we proposed two new methods of crossover for CGP. In our evaluation part of our experiments, the use of the subgraphand block crossover resulted in a significantly smaller number of generations until the CGP algorithm terminated successfully, in a better fitness value, and better convergence behavior. The experiments on our test problems indicate that the subgraph and the block crossover technique may be beneficial for the search performance of CGP. Moreover, we have demonstrated the potential of both methods for a diverse set of problems, including different functions and types of fitness. Our preliminary comparison to the arithmetic crossover technique indicates a significantly better performance for the subgraph and the block crossover on our tested problems.

In addition to the proposal of two new crossover techniques, a comparative study on crossover in CGP has been proposed. We have performed a comparative study with two evolutionary methods that use only mutation, and four other crossover operators. Beside the block crossover, simple one-point crossover, arithmetic crossover, used in the field of real-valued Genetic Algorithms, and Subgraph Crossover that recombines parts of the parent chromosome phenotypes were included.

We performed a comparison on eight selected tasks from the areas of Boolean function design and symbolic regression. We have used meta-evolution to determine the most important evolutionary parameters and find common values for the parameters of lower importance.

Next, we have performed a series of parameter sweeps, to determine the settings most suitable for every type of crossover and every task, and performed a comparison. Finally, we have performed a non-parametric statistical test to prove our hypothesis false and shown that the $(1 + \lambda)$-CGP is significantly outperformed by all other approaches, when designing 1-resilient Boolean functions.

The results show, that it is possible for crossover operators to outperform the standard $(1 + \lambda)$ strategy. However, if both methods have their parameters fine-tuned, the $(1+\lambda)$ strategy often remains as the overall best strategy. The question of finding a universal crossover operator is CGP, therefore, remains open.

This chapter opens a new perspective on comparative studies on the use of crossover in CGP and its challenges. The experiments with meta-evolution in CGP have shown that it is difficult to obtain well-performing parameter settings for crossover algorithms in CGP.

The following concluding remarks can be formulated on the basis of the results of our experiments:

- It is possible for crossover operators to outperform the standard $(1+\lambda)$-strategy in CGP
- It is a difficult task to determine the optimal parameter setting for crossover based algorithms since the dimensionalty of the parametrization evaluation task is generally higher (e.g. more parameters have to be tuned)
- A larger comparative study is needed which is oriented with the parameter tuning approach in Chapter 5

8 Advanced Mutation Operators

8.1 Introduction

In contrast to tree-based GP for which a broad range of advanced crossover and mutation techniques have been introduced and investigated; the state of knowledge of advanced mutation techniques in CGP appears to be relatively weak. In standard tree-based GP, the simultaneous use of multiple types of mutation has been found beneficial by Kraft et al. [69] and Angeline [1]. In this chapter, we propose two phenotypic mutations for CGP and take a step toward advanced phenotypic mutations in CGP. Furthermore, we present comprehensive experiments with Boolean function and symbolic regression problems to demonstrate that our proposed mutation can be beneficial for the use of CGP.

8.2 Related Work

8.2.1 Advanced Mutation Techniques in standard CGP

For an investigation of the length bias and search limitation of CGP, a modified version of the point mutation has been introduced by Goldman and Punch [32]. The modified point mutation exactly mutates one active gene. This so-called single active-gene mutation strategy (SAGMS) has been found beneficial for the search performance of CGP. The SAGMS can be seen as a form of phenotypic genetic operator since it respects only active function genes in the genotype which are an active part of the corresponding phenotype. Later work by Manfrini et al. [89] extended SAGMS to the so-called *Biased Single-Active Mutation*, which is generally based on the idea of analyzing the genotypic behavior of the during the evolutionary run for a given set of problems. With this analysis, a bias is created in order to help the direct gene mutation when applied to other problems. For every time, when the fitness of the offspring has better fitness as its parent, the mutation is considered beneficial. Every time when a beneficial mutation occurs, it is examined if this mutation occurred on a executing function of the parent. In this case, the beneficial function transition from the function in the parent genome to the function in the offspring genome is stored. With the storage of beneficial function transitions, a frequency table of all transitions is created, which is used to create a probability distribution of function transitions. The mutation operator was proposed for digital combinational logic circuit design. The experiments of Manfrini et al. demonstrated that the proposed mutation performed better or equivalent as the traditional point mutation. To reduce the stalling effect in CGP and to improve the efficiency of the CGP al-

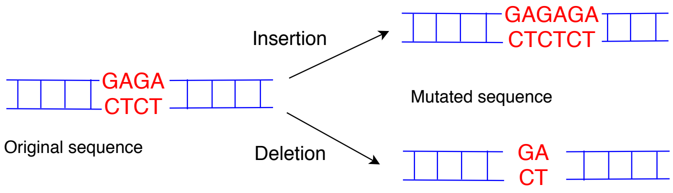


Figure 8.1: Deletions and insertions of nucleotides.

gorithm, Ni et al. [108] introduced an Orthogonal Neighbourhood Mutation (ONM) operator.The results of the experiments demonstrated that the ONM operator can reduce the stalling effect in CGP and the algorithm improves more quickly.
The proposed mutations for CGP are inspired by biological evolution in which extra base pairs are inserted into a new place in the DNA or in which a section of DNA is deleted. Figure 8.1 exemplifies the insertion and deletion mutation on the DNA sequence. Related to CGP, we adopt these so-called *frameshift mutations* by activating and deactivating randomly chosen function nodes. The activation and deactivation of the nodes are done by adjusting the connection genes of neighborhood nodes. Both mutation techniques work similarly as the single active-gene mutation strategy. The state of exactly one function node in the genome is changed. Since these forms of mutation can elicit strong changes in the behavior of the individuals, we apply an *insertion rate* and a *deletion rate* for every offspring. Based on these mutation rates, the decision is made whether the mutations are performed on the genome of an individual. The insertion and deletion mutation techniques work independently from each other, which means that both mutations can be performed on the genome of the individual in the breeding procedure of one generation. If the consideration of a minimum or a maximum number of function nodes is necessary for all individuals in the population, the algorithms can be parameterized with maximum and minimum numbers. We will explain both mutation techniques in detail in the following two subsections. For both mutation techniques, we determine the active and passive function nodes of the respective individual before the mutation procedure.

8.3 The Insertion Mutation Technique

When a genome is selected for the insertion mutation, one inactive function node becomes active. If all function nodes are already active or the number of active function nodes excels a defined maximum, the mutation is rejected. If an individual is suitable for the insertion mutation, we randomly select one inactive function node. After selection, we have to distinguish three cases:

1. **The selected inactive node has a following active function node**

   In this context, the term *following function node* means that the node number of an active function node is higher than the function number of the randomly selected node. If the selected node has a following active function node, we copy the connection genes of the next active node to the selected inactive node. Afterward, we adjust one randomly selected connection gene of the following active node to the selected inactive node. In this way, the selected inactive node will be respected by the backward search and consequently becomes active. No further steps are required for the previously active function node since all other active function nodes remain active due to the copying of the connection genes.

2. **The selected inactive node has a previous active function node and no following active function node**

   In this context, the term *previous active function node* means that the node number of an active function node is smaller than the function number of the randomly selected node. If the selected node has a previous active function node and no following active node, at least one output node is connected with the previous active function node. In this case, we adjust all output nodes which are connected with the previously active function node to the selected inactive node. Afterward, we adjust one connection gene of the selected node to the previously active function node. The other connection genes are randomly connected to previous active function or input nodes. In this way, the selected inactive node becomes active, and the other inactive function nodes remain inactive.

3. **The selected inactive node has no previous or following active function node**

   If the selected inactive node has no previous or following active function node, the individual has no active function nodes. Consequently, the output nodes are directly connected with an input node. If this is the case, we adjust at least one output node to the selected inactive node. Afterward, we randomly connect the connection genes of the selected inactive node to input nodes. In this way, the selected node becomes active, and other function nodes remain inactive.

Definition 8.1 (Insertion Mutation)**.** *The insertion mutation is a variation operator for CGP which selects and activates one inactive function node of a cartesian genetic program, where the inactive node is selected by chance.*

The three cases are illustrated in Figure 8.2. A defintion of the insertion mutation is given in Definition 8.1. Figure 8.3 exemplifies the insertion technique. In the figure, the genotype is grouped into a number of genes that represent the function and output nodes. Moreover, active function nodes are highlighted in solid boxes, and inactive nodes are shown in dashed boxes. The selected active or inactive nodes are highlighted in red. As visible, one inactive node is selected for activation. The connection genes in the genotype are adjusted to activate the selected function node

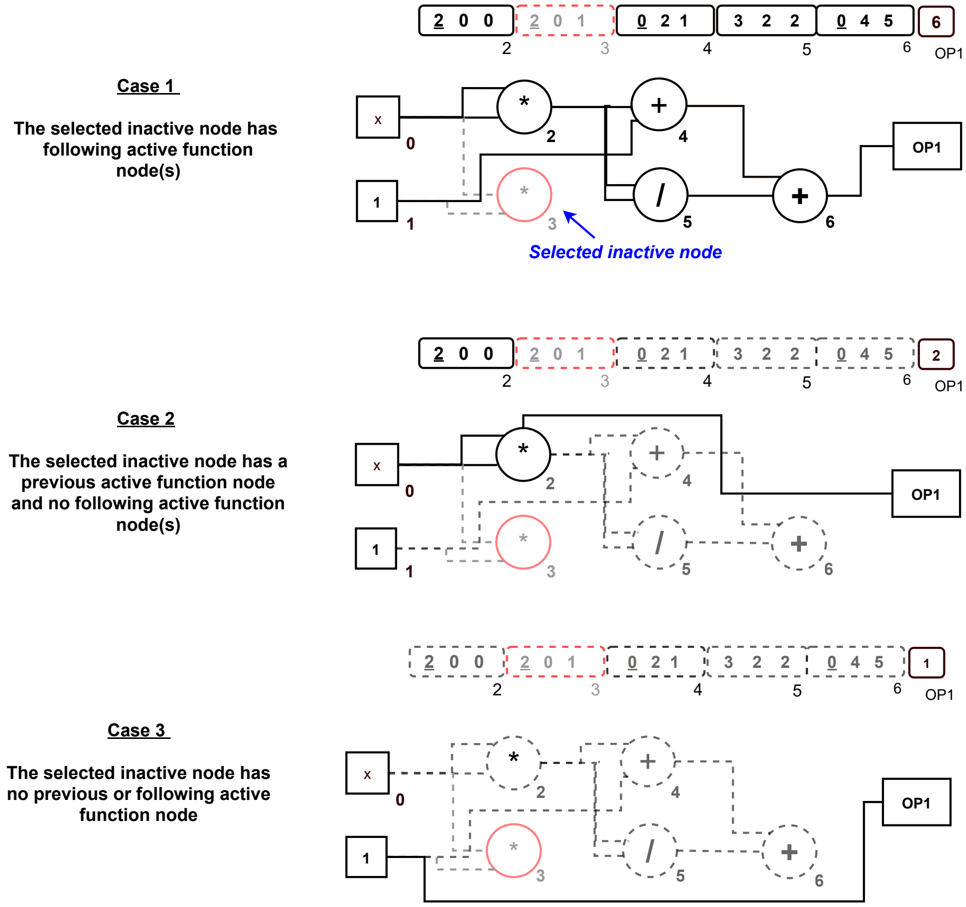

Case 1
The selected inactive node has following active function node(s)
2 0 0 | 2 0 1 | 0 2 1 | 3 2 2 | 0 4 5 | 6
2 3 4 5 6 OP1
x 0
1 1
* 2
* 3
+ 4
/ 5
+ 6
OP1
Selected inactive node
Case 2
The selected inactive node has a previous active function node and no following active function node(s)
2 0 0 | 2 0 1 | 0 2 1 | 3 2 2 | 0 4 5 | 2
2 3 4 5 6 OP1
Case 3
The selected inactive node has no previous or following active function node
2 0 0 | 2 0 1 | 0 2 1 | 3 2 2 | 0 4 5 | 1
2 3 4 5 6 OP1

Figure 8.2: The three cases which have to be distinguished for the insertion mutation procedure.

in the phenotype. The procedure is described in more detail in Algorithm 8.5. The corresponding subfunctions of the procedure are described in Algorithms 7.1, 8.1, 8.2, 8.3 and 8.4.

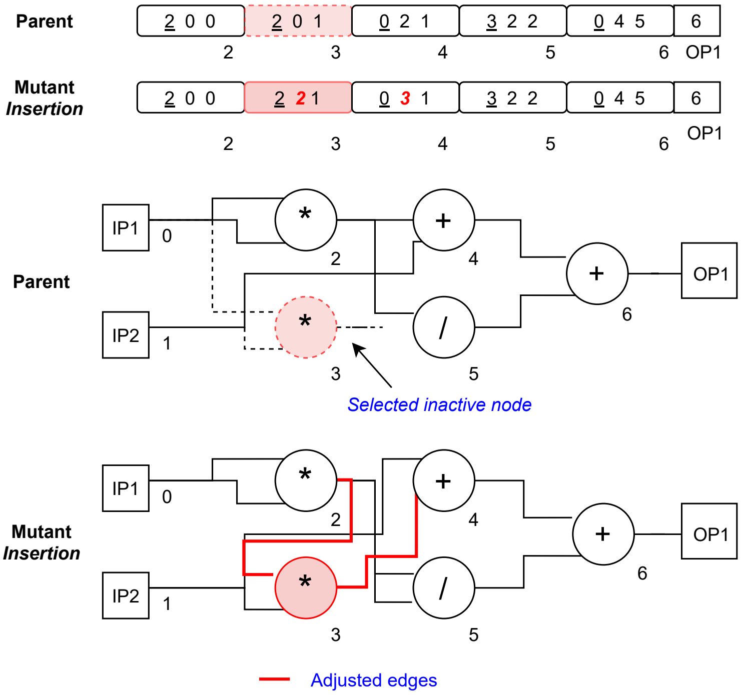


Figure 8.3: The proposed insertion mutation technique.

Algorithm 8.1 Insertion mutation: Function for the determination of the following active function node (FAFN)

```
Arguments
$N_\mathrm{A}$: Sorted list containing the active node numbers
$n_\mathrm{p}$: A previously selected number of an inactive/passive node
Return
$n_\mathrm{c}$: Determined following active function node number
$i_\mathrm{c}$: Corresponding index in $N_\mathrm{A}$
function DetermineFAFN($N_\mathrm{A}$, $n_\mathrm{p}$)
    $j \leftarrow 0$                                   ▷ Initialize the loop counter
    $n_c \leftarrow N_\mathrm{A}[0]$                   ▷ Store the first active node in $n_\mathrm{c}$
    ▷ While the given passive node number is smaller than the current active node number
    while $n_c < n_\mathrm{p}$ do
        $j \leftarrow j + 1$                           ▷ Increase the loop counter
        $i_c \leftarrow j$                             ▷ Store the index in $i_\mathrm{c}$
        $n_c \leftarrow N_\mathrm{A}[j]$               ▷ Determine the current active function node
    end while
    ▷ Return the the following active function node and the corresponding index in $N_\mathrm{A}$
    return $n_c, i_c$
end function
```

Algorithm 8.2 Insertion mutation: Function for the determination of the starting position in the genome of a given node number

Arguments
n: Function node number
i: Number of inputs
a: Maximum arity size
Return
p: Starting position of n

```
function PositionFromNodeNumber($n$, $i$, $a$)
    ▷ The genome position is calculated with the node number, number of inputs and the maximum arity
    $p \leftarrow (n - i) + (a + 1)$
    return $p$                                        ▷ Return the position
end function
```

Algorithm 8.3 Insertion mutation: Function for the adjustment of the output nodes

Arguments
G: Genome of an individual
O: List with output nodes
N_A: List with active function node numbers
n_r: Randomly selected inactive function node
s_a: Number of active function nodes
Return
G: Genome with adjusted outputs

```
function AdjustOutputs($G$,$O$, $N_\mathrm{A}$, $n_\mathrm{r}$, $s_\mathrm{a}$ )
    $n_l \leftarrow N_\mathrm{A}[s_\mathrm{a} - 1]$          ▷ Save the last active function node in $n_\mathrm{l}$
    $s_g \leftarrow |G|$                                     ▷ Determine the size of $G$
    $j \leftarrow 1$                                         ▷ Initialize loop counter
    while $j \leq |O|$ do                                    ▷ Iterate over the number of outputs
        if $G[s_g - j] = n_l$ then        ▷ If an output is connected to the last active function node
            $G[s_g - j] \leftarrow n_\mathrm{r}$   ▷ Rewire the output to the randomly selected inactive node $n_\mathrm{r}$
            break                         ▷ The node is now active, consequently abort the iteration
        end if
        $j \leftarrow j + 1$                                 ▷ Increase the loop counter
    end while
    return $G$                                               ▷ Return the Genome
end function
```

Algorithm 8.4 Insertion mutation: Function for the adjustments of the connection genes of a particular function node

```
Arguments
$G$: Genome of an individual
$N_\mathrm{A}$: List with active function node numbers
$I$: List with input nodes
$n_\mathrm{r}$: Randomly selected inactive function node
$p_\mathrm{r}$: Position of $n_\mathrm{r}$
$a$: Maximum arity size
Return
$G$: Genome with adjusted connection genes

function AdjustConnectionGenes($G$, $N_\mathrm{A}$, $I$, $n_\mathrm{r}$, $p_\mathrm{r}$, $a$ )
    $j \leftarrow 1$                                   ▷ Initialize loop counter
    $n_p \leftarrow N_\mathrm{A}$.last()               ▷ Store the previous active function node of $n_\mathrm{r}$ (last active function node)
    while $j \leq a$ do                                ▷ Iterate over the maximum arity
        $p_c \leftarrow p_\mathrm{r} + j$              ▷ Calculate the position depending on the iteration
        if $j = 1$ then                                ▷ Check for the first connection gene
            $G[p_c] \leftarrow n_p$                    ▷ Rewire this connection to the previous active function node
        else
            ▷ The other connectios are rewired to randomly selected input or previous active function nodes
            $G[p_c] \leftarrow$ RandomNodeNumber($I$,$N_\mathrm{A}$, $n_p$)    ▷ Algorithm 7.1
        end if
    end while
    return $G$                                         ▷ Return the genome
end function
```

Algorithm 8.5 Insertion mutation: The function for the main mutation procedure

```
    Arguments
    $N_\mathrm{A}$: Sorted list of active node numbers, $N_\mathrm{I}$: Sorted list of inactive node numbers
    $G$: Genome of an individual, $a$: Maximum arity size
    $O$: List with output nodes, $I$: List with input nodes
    $a$: The maximum arity
    Return
    $G$: The mutated genome
function Insertion($N_\mathrm{A}$,$N_\mathrm{I}$, $G$, $a$, $O$, $I$)
    $n_r \leftarrow$ RandomNodeNumber($N_\mathrm{I}$)    ▷ Get an inactive random node number (Alg. 7.1)
    ▷ Get the genome position of node $n_\mathrm{r}$ (Alg. 8.2)
    $p_r \leftarrow$ PositionFromNodeNumber($n_\mathrm{r}$, $i$, $a$)

    $s_a \leftarrow |N_\mathrm{A}|$    ▷ Determine the size of $N_\mathrm{A}$
    $s_i \leftarrow |N_\mathrm{I}|$    ▷ Determine the size of $N_\mathrm{I}$
    if $s_i = 0$ then    ▷ If all nodes are active, abort the mutation procedure
        return
    end if
    if $s_\mathrm{a} > 0$ then    ▷ First two cases, at least one node is active
        if $n_\mathrm{r} < N_\mathrm{A}[s_a - 1]$ then ▷ Case 1: Node $n_\mathrm{r}$ has following active function node(s)
            ▷ Determine the first following active function node (FAFN) (Alg. 8.1)
            $n_f \leftarrow$ DetermineFAFN($N_\mathrm{A}$, $n_r$)
            ▷ Get the genome position of node $n_f$ (Alg. 8.2)
            $p_f \leftarrow$ PositionFromNodeNumber($n_f$, $i$, $a$)
            $j \leftarrow 1$    ▷ Initialize the loop counter
            while $j \leq a$ do    ▷ Loop over the maximum arity
                $p_c \leftarrow p_r + j$    ▷ Store the current gene position of the interation in $p_c$
                $G[p_c] \leftarrow G[p_f + j]$    ▷ Transfer the current gene from the node $n_f$ to $n_r$
                $j \leftarrow j + 1$    ▷ Increase the loop counter
            end while
            ▷ Choose a connection gene for node $n_f$ by random
            $r_i \leftarrow$ RandomInteger$(0, a - 1)$
            $G[p_f + r_i + 1] \leftarrow n_r$    ▷ Rewire the connection to node $n_r$, $n_r$ will become active

            ▷ Case 2: Node $n_r$ has a previous active function node and no following ones
        else if $n_r > N_\mathrm{A}[s_a - 1]$ then
            $G \leftarrow$ AdjustOutputs($G$, $O$, $N_\mathrm{A}$, $n_r$, $s_a$)    ▷ Adjustment of the outputs (Alg. 8.3)
            ▷ Adjust the connection genes (Alg. 8.4)
            $p_r \leftarrow$ PositionFromNodeNumber($n_r$, $i$, $a$)
            $G \leftarrow$ AdjustConnectionGenes($G$, $N_\mathrm{A}$, $I$, $n_r$, $p_r$, $a$)
        end if
    else    ▷ Case 3: Node $n_r$ has no previous or following active function node
        $s_g \leftarrow |G|$    ▷ Determine the size of $G$
        $G[s_g - 1] \leftarrow n_r$    ▷ Connect one output to the selected inactive node
        $j \leftarrow 1$    ▷ Initialize loop counter
        while $j \leq a$ do    ▷ Iterate over the maximum arity
            $p_c \leftarrow p_r + j$ ▷ Calculate and save the position of the j-th connection gene of node $n_r$
            ▷ Connect the connection gene with a randomly selected input
            $G[p_c] \leftarrow$ RandomNodeNumber($I$)
        end while
    end if
    return $G$    ▷ Return the mutated genome
end function
```

8.4 The Deletion Mutation Technique

In contrast to the insertion mutation technique, when a genome is selected for deletion mutation, one active node becomes inactive. If all function nodes are inactive or the number of active function nodes is smaller than a defined minimum, the mutation is rejected. If an individual is suitable for the deletion mutation, we select the first active function node of the individual.
The deletion mutation procedure is then done by performing the following steps:

a **Adjust the connection genes of all following active function nodes**

The connection genes of all following active function nodes which are connected with the selected active function node are randomly adjusted to other active function or input nodes.

b **Adjust the outputs nodes**

All output nodes which are connected with the selected active function nodes are randomly adjusted to other active function or input nodes.

After performing the adjustment of connection genes and output nodes, the selected active function node becomes inactive. A defintion of the delection mutation is given in Definition 8.2.

Definition 8.2 (Deletion Mutation)**.** *The deletion mutation is a variation operator for CGP which deactivates the first active function node of a cartesian genetic program.*

Figure 8.4 illustrates the deletion mutation where one active node becomes inactive by adjusting the respective connection genes. The algorithmic procedure is described in more detail in Algorithm 8.7. The corresponding subfunctions are described in Algorithms 7.1 and 8.6.

8.5 Experiments

8.5.1 Experimental Setup

We performed experiments with Boolean function and symbolic regression problems. To evaluate the search performance of the insertion and deletion of mutation techniques, we measured the number of fitness evaluations until the CGP algorithm terminated and the best fitness which has been found. In addition to the mean values of the measurements, we also calculated the standard deviation (SD) and the standard error of the mean (SEM), the median and the first and third quartile. We performed 100 independent runs with different random seeds and applied the well known $(1+4)$-CGP algorithm for all experiments. Moreover, we used the standard

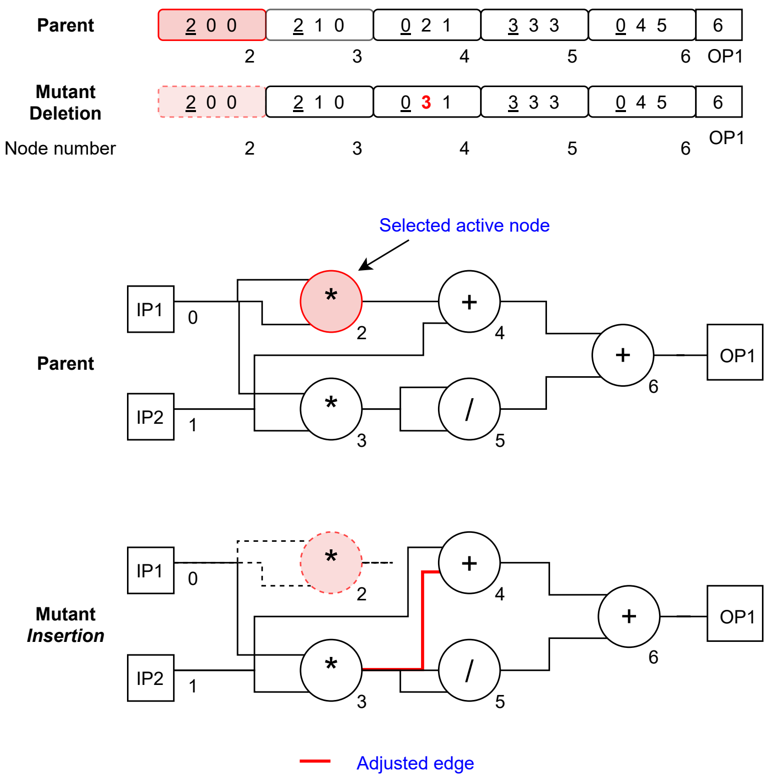


Figure 8.4: The proposed deletion mutation technique

Algorithm 8.6 Deletion Mutation Technique: Subfunction for the decoding of the gene type

```
    Arguments
    $p$: Position of the gene
    $g$: Value of the gene
    $n$: Number of function nodes
    $a$: Maximum arity size
    Return
    $g_c$, $g_f$ or $g_o$: Type of the gene at position $p$

$g_c \leftarrow 0$                                   ▷ Integer to represent a connection gene
$g_f \leftarrow 1$                                     ▷ Integer to represent a function gene
$g_o \leftarrow 2$                                       ▷ Integer to represent a output gene
function DecodeGenotype($p$, $n$, $a$)
    if $p \geq n \cdot (a+1)$ then    ▷ If p exceeds the maximum index of function/connection genes
        return $g_o$                                              ▷ It must be an output gene
        ▷ Each node is represented with a+1 genes, so we can use the modulo operator to check if it is a function gene
    else if $p \ \% \ (a+1) = 0$ then
        return $g_f$
    else                                          ▷ Last possible case: Its a connection gene
        return $g_c$
    end if
end function
```

Algorithm 8.7 Deletion Mutation Technique: The main mutation procedure

```
Arguments
$N_{\mathrm{A}}$: Sorted list of active function nodes , $G$: Genome of an individual
$a$: Maximum arity size, $O$: List of outputs, $I$: List of inputs
Return
$G$: The mutated genome

$g_c \leftarrow 0$                                      ▷ Integer to represent a connection gene
$g_f \leftarrow 1$                                      ▷ Integer to represent a function gene
$g_o \leftarrow 2$                                      ▷ Integer to represent a output gene
function Deletion($N_{\mathrm{A}}$,$N_{\mathrm{I}}$, $G$, $a$, $O$, $I$)
    $s_a \leftarrow |N_{\mathrm{A}}|$                   ▷ Determine the size of $N_{\mathrm{A}}$
    $s_i \leftarrow |N_{\mathrm{I}}|$                   ▷ Determine the size of $N_{\mathrm{I}}$
    $s_n \leftarrow s_a + s_i$                          ▷ Calculate the total number of function nodes
    if $s_a = 0$ then                                   ▷ If no function node is active, abort the mutation procedure
        return
    end if
    $n_0 \leftarrow N_{\mathrm{A}}[0]$                  ▷ Store the first active function node in $n_0$
    $N_{\mathrm{A}}$.remove($n_0$)                      ▷ Remove the first active function node from the list
    $j \leftarrow 0$
    while $j < |G|$ do                                  ▷ Iterate over the genome
        ▷ Determine and store the type of the gene (Alg. 8.6)
        $g \leftarrow$DecodeGenotype($G[j]$,$s_n$,$a$)
        if $g = g_c$ then                               ▷ If it is a connection gene
            $n_c \leftarrow$NodeNumber($j$,$G$)         ▷ Determine the current node number
            ▷ If the node is active and linked with the first active function node
            if $N_{\mathrm{A}}$.contains($n_c$) and $G[j] = n_0$ then
                ▷ Rewire the connection gene with a randomly selected input or active function node
                $G[j] \leftarrow$ RandomNodeNumber($I$, $N_A$, $n_c - 1$)      ▷ Alg. 7.1
            end if
            ▷ If a output is connected to the deleted first active function node
        else if $g = g_o$ and $G[j] = n_0$ then
            ▷ Rewire the output gene with a randomly selected active function or input node (Alg. 7.1)
            $G[j] \leftarrow$ RandomNodeNumber($I$, $N_A$)
        end if
        $j \leftarrow j + 1$                            ▷ Increase loop counter
    end while
    return $G$                                          ▷ Return the mutated genome
end function
```

Problem	**Number of Inputs**	**Number of Outputs**
Parity-3	3	1
Parity-4	4	1
Parity-5	5	1
Parity-6	6	1
Parity-7	7	1
Adder 1-Bit	3	2
Adder 2-Bit	5	3
Adder 3-Bit	7	4
Multiplier 2-Bit	4	4
Multiplier 3-Bit	6	6
Demultiplexer 3:8-Bit	3	8
Comparator 4x1-Bit	4	18

Table 8.1: Boolean function problems for the search performance evaluation.

Property	**Value**
μ	1
λ	4
Number of nodes	100
Maximum generations	20000000
Function set	AND, OR, NAND, NOR
Point mutation rate	4%

Table 8.2: Configuration of the $1 + 4$-CGP algorithm.

CGP point mutation operator in combination with the insertion and deletion mutations. We considered minimization problems in all experiments which are explained in the respective subsection. To assess the significance of our results, we used the Mann-Whitney-U-Test. The mean values are denoted $a^{\dagger}$ if the p-value is less than the significance level 0.05 and $a^{\ddagger}$ if the p-value is less than the significance level 0.01 compared to the use of the point mutation as the sole genetic operator. The levels back parameter l was set to ∞.

8.5.2 Search Performance Evaluation

Boolean Function Problems

To evaluate the search performance of the insertion and deletion mutation techniques, we chose the five Even-Parity problems with $n = 3$, 4, 5, 6, and 7 Boolean inputs. The goal was to find a program that produces the value of the Boolean even parity depending on the n independent inputs. The fitness was represented by the number of fitness cases for which the candidate solution failed to generate the correct value of the even parity function.

Problem	**Point mutation rate** [%]	**Insertion rate** [%]	**Deletion rate** [%]
Parity-3	2.5	40	25
Parity-4	1.5	7.5	5
Parity-5	1	8	2
Parity-6	1	6	4
Parity-7	1	6	3
Adder 1-Bit	2	5	5
Adder 2-Bit	1	10	10
Adder 3-Bit	1	5	5
Multiplier 2-Bit	2	5	5
Multiplier 3-Bit	1	6	3
Demultiplexer 3:8-Bit	2	10	10
Comparator 4x1-Bit	1	5	5

Table 8.3: Insertion and deletion rates for the $(1+4)$-CGP-ID algorithm.

Since former work by White et al. [160] outlined that this problem type was excessively used and investigated in the past, we also evaluated multiple output problems as the digital adder, multiplier, and demultiplexer. These types of problems differ markedly from the parity problems, and the 3-Bit digital multiplier has been proposed as a suitable alternative. As a result, we receive a diverse set of problems in this problem domain. The set of benchmark problems with the corresponding number of inputs and outputs is shown in Table 8.1. To evaluate the fitness of the individuals on the multiple output problems, we defined the fitness value of an individual as the number of different bits to the corresponding truth table. To find performant configurations for the insertion and deletion mutation rates, we used automated parameter tuning. The evolved configurations are shown in Table 8.3.
We compared the $(1+4)$-CGP algorithm to our modified $(1+4)$-CGP algorithm equipped with the insertion and deletion mutation techniques. Our modified $(1+4)$-CGP is denoted as $(1+4)$-CGP-ID. The number of function nodes was set to 100 for all tested problems. Following conventional wisdom for CGP, we use a point mutation rate of 4% for the traditional $(1+4)$-CGP algorithm. The algorithm configuration of the $(1+4)$-CGP algorithm is shown in Table 8.2. We performed the runtime measurement on a computer with a Intel(R) Core(TM) i7 CPU 930 with 2.80 GHz and 24 GB of RAM.
Table 8.4 presents the results of our search performance evaluation, which show a reduced number of generations until the termination criterion triggers for the $(1+4)$-CGP-ID algorithm. The results also show that when the $(1+4)$-CGP-ID is used on more complex Boolean function problems, the mean runtime of the algorithm is also clearly reduced. Figure 8.5 provides boxplots for all tested problems of the search performance evaluation.

Symbolic Regression

To evaluate the search performance in the symbolic regression domain, we chose eleven symbolic regression problems from the work of McDermott et al. [90] for

Problem	**Algorithm**	**Mean** **Fitness Evaluation**	**SD**	**SEM**	**1Q**	**Median**	**3Q**	**Mean** **Runtime**
Parity-3	(1 + 4)-CGP	4917	4926	±493	1695	3412	5598	0.20 s
	(1 + 4)-CGP-ID	**2700**‡	**2173**	**±217**	**1370**	**1928**	**3358**	**0.17** s
Parity-4	(1 + 4)-CGP	43895	43013	±4301	18125	29398	57968	1.78 s
	(1 + 4)-CGP-ID	**14381**‡	**9905**	**±991**	**8948**	**11928**	**19948**	**1.04** s
Parity-5	(1 + 4)-CGP	194727	148386	±14839	83304	168996	249993	12.47 s
	(1 + 4)-CGP-ID	**45349**‡	**28257**	**±2826**	**25735**	**34622**	**53923**	**6.99** s
Parity-6	(1 + 4)-CGP	746627	512510	±51250	371794	617932	937638	112.35 s
	(1 + 4)-CGP-ID	**105331**‡	**52171**	**±5217**	**65445**	**92466**	**139067**	**38.22** s
Parity-7	(1 + 4)-CGP	3074853	3146951	±314695	1341520	2231156	3696237	976.68 s
	(1 + 4)-CGP-ID	**283856**‡	**177515**	**±17751**	**177610**	**238426**	**325776**	**181.09** s
Adder 1-Bit	(1 + 4)-CGP	9364	8002	±800	3183	7550	12413	0.23 s
	(1 + 4)-CGP-ID	**8080**†	**7360**	**±736**	**3448**	**5876**	**10254**	0.23 s
Adder 2-Bit	(1 + 4)-CGP	274734	262394	±26239	113622	188212	341853	4.98 s
	(1 + 4)-CGP-ID	**113744**‡	**88022**	**±8802**	**56379**	**84258**	**140745**	**3.53** s
Adder 3-Bit	(1 + 4)-CGP	4068492	3567764	±356776	1802712	3092538	4745253	90.93 s
	(1 + 4)-CGP-ID	**846075**‡	**885420**	**±88542**	**373149**	**584198**	**979748**	**36.39**] s
Multiplier 2-Bit	(1 + 4)-CGP	24645	33364	±3336	6499	14108	26148	0.48 s
	(1 + 4)-CGP-ID	**21539**‡	**33170**	**±3317**	**6372**	**10196**	**23753**	**0.47** s
Multiplier 3-Bit	(1 + 4)-CGP	757523	522412	±52241	402333	685390	958647	14.60 s
	(1 + 4)-CGP-ID	**354118**‡	**337590**	**±33759**	**142446**	**250396**	**465565**	**9.49**s
Demultiplexer 3:8-Bit	(1 + 4)-CGP	23432	13546	±1355	15258	199918	26750	0.60 s
	(1 + 4)-CGP-ID	**15523**‡	**8994**	**±899**	**8954**	**13704**	**19657**	**0.53** s
Comparator 4x1-Bit	(1 + 4)-CGP	2628085	1848923	±184892	1528983	2056080	2918599	91.06 s
	(1 + 4)-CGP-ID	**338019**‡	**208523**	**±20852**	**180908**	**272924**	**461282**	**14.65** s

Table 8.4: Results of the search performance evaluation for the tested Boolean function problems evaluated by the number of fitness evaluations to termination.

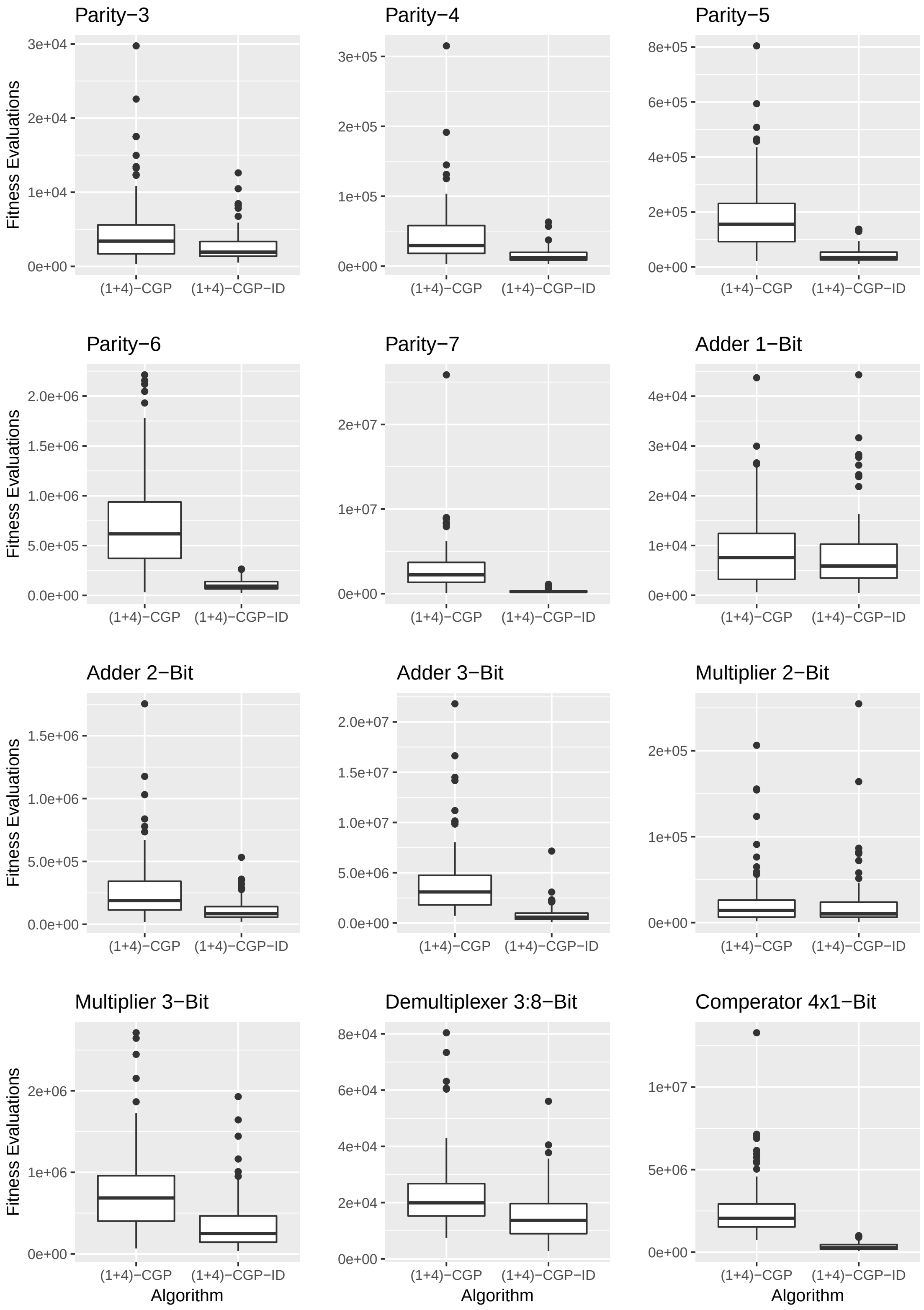


Figure 8.5: Boxplots for the results of the search performance evaluationv.

better GP benchmarks. The functions of the problems are shown in Table 8.6. A training data set U$[a, b, c]$ refers to c uniform random samples drawn from a to b inclusive and E$[a, b, c]$ refers to a grid of points evenly spaced with an interval of c, from a to b inclusive. The respective constants and function sets are shown in Table 8.7. We included the problems Keijzer-6, Nguyen-7, Pagie-1, Vladislavleva-4 and Korns-12, which have been recommended by White et al. [160] as a set of significant problems with different reputations. According to White et al. [160] the Keijzer-6 and Vladislavleva-4 problems require extrapolation, not just interpolation. The Korns-12 problem is the only problem which could not be solved in [64] even when several specialized techniques were applied. According to [160] the dataset is specified, consisting of 5 variables, but only 2 affect the output. In this way, the problems challenges the ability of systems to discard unimportant variables. The authors of [154] state that by attempting to solve the Vladislavleva-4 problem their system appears to have most difficulties in discovering the simple and harmonious input–output relationship. The Pagie-1 problem has a reputation for difficulty [37, 112] despite being smooth and scalable. White et al. [160] also stated that the same is true for the Vladislavleva-4 problem. The Nguyen-7 problem features different basic functions from the others.

The fitness of the individuals was represented by a cost function value. The cost function was defined by the sum of the absolute difference between the true function values and the values of an evaluated individual.

When the sum of absolute differences becomes less than 0.01, the algorithm is classified as converged. All problems were evaluated with the *best-fitness-of-run* method. We measured the best fitness value after a budget of 10000 generations. Additionally, we evaluated the simpler symbolic regression problems Koza 1,2 & 3 with the *fitness-evaluation-to-termination* method. For this purpose we used a smaller function set consisting of the four basic arithmetic functions $+, -$, $*$ and $/$. We defined a maximum number of 10^6 fitness evaluations for these three experiments. The reason for our choice of these three problems is the fact that we can find an ideal solution more likely on average than the other more complex benchmark problems, which require a higher amount of fitness evaluations to find an ideal solution. To find performant configurations for the insertion and deletion mutation rates, we tuned the respective parameters manually. For all tested problems, we observed that a point mutation rate of 4% seems to be a good choice for the use of the insertion and deletion mutations. The determined configurations are shown in Table 8.5.

Table 8.8 and 8.9 present the results of our search performance evaluation in the symbolic regression domain, which show a better (smaller) mean fitness value of run and a reduced number of fitness evaluations until the termination criterion triggers for the $(1+4)$-CGP-ID algorithm. Figure 8.6 and 8.7 provide boxplots for all tested problems of the search performance evaluation.

Problem	Point mutation rate [%]	Insertion rate [%]	Deletion rate [%]
Koza-1	4	5	5
Koza-2	4	5	5
Koza-3	4	5	7.5
Nguyen-4	4	5	5
Nguyen-5	4	5	5
Nguyen-7	4	7.5	5
Keijzer-6	4	10	5
Pagie-1	4	5	5
Vladislavleva-4	1	5	5
Korns-12	1	10	5

Table 8.5: Insertion and deletion rates for the (1 + 4)-CGP-ID algorithm for the tested symbolic function problems.

Table 8.6: List of symbolic regression problems for the search performance evaluations.

Problem	Objective Function	Vars	Training Set	Function Set
Koza-1	$x^4 + x^3 + x^2 + x$	1	U[-1,1,20]	Koza
Koza-2	$x^5 - 2x^3 + x$	1	U[-1,1,20]	Koza
Koza-3	$x^6 - 2x^4 + x^2$	1	U[-1,1,20]	Koza
Nguyen-4	$x^6 + x^5 + x^4 + x^3 + x^2 + x$	1	U[-1,1,20]	Koza
Nguyen-5	$\sin(x^2)\cos(x) - 1$	1	U[-1,1,20]	Koza
Nguyen-6	$\sin(x) + \sin(x + x^2)$	1	U[-1,1,20]	Koza
Nguyen-7	$\ln(x + 1) + \ln(x^2 + 1)$	1	U[0,2,20]	Koza
Keijzer-6	$\sum_i^x 1/i$	1	E[1,50,1]	Keijzer
Pagie-1	$1/(1 + x^{-4}) + 1/(1 + y^{-4})$	2	E[-5,5,0.4]	Koza
Vladislavleva-4	$\frac{10}{5+(x-3)^2+(y-3)^2+(z-3)^2+(v-3)^2+(w-3)^2}$	5	U[0.05,6.05,1024]	Vladislavleva-A
Korns-12	$2 - 2.1\cos(9.8x)\sin(1.3w)$	5	U[-50, 50, 10000]	Korns

Table 8.7: Function sets for the set of symbolic regression problems.

Name	Functions	Constants (ERC)
Koza	$+ \quad - \quad * \quad / \quad \sin \quad \cos \quad e^n \quad \ln(\lvert n \rvert)$	Constant input with a value of 1
Keijzer	$+ \quad * \quad \frac{1}{n} \quad -n \quad \sqrt{n}$	Random value from $N(\mu = 0, \sigma = 5)$
Vladislavleva-A	$+ \quad - \quad * \quad / \quad n^2$	$n^\epsilon \quad n + \epsilon \quad n\epsilon$
Korns	$+ \quad - \quad * \quad / \quad \sin \quad \cos \quad e^n \quad \ln(\lvert n \rvert)$ $n^2 \quad n^3 \quad \tan \quad \tanh \quad \sqrt{n}$	Random finite 64-bit IEEE 754 double-precision number

Problem	**Algorithm**	**Mean Best Fitness of Run**	**SD**	**SEM**	**1Q**	**Median**	**3Q**
Koza-1	(1 + 4)-CGP	0.33	0.36	±0.03	0.13	0.23	0.38
	(1 + 4)-CGP-ID	**0.18**‡	**0.15**	**±0.01**	**0.06**	**0.15**	**0.26**
Koza-2	(1 + 4)-CGP	0.16	0.142	±0.01	0.04	0.13	0.23
	(1 + 4)-CGP-ID	**0.11**‡	**0.12**	**±0.01**	**0.03**	**0.07**	**0.17**
Koza-3	(1 + 4)-CGP	0.09	0.10	±0.01	0.03	0.06	0.13
	(1 + 4)-CGP-ID	**0.06**†	**0.05**	**±0.00**	**0.02**	**0.04**	**0.08**
Nguyen-4	(1 + 4)-CGP	0.36	0.30	±0.03	0.18	0.30	0.48
	(1 + 4)-CGP-ID	**0.24**‡	**0.19**	**±0.02**	**0.11**	**0.18**	**0.34**
Nguyen-5	(1 + 4)-CGP	0.21	0.18	±0.02	0.07	0.15	0.31
	(1 + 4)-CGP-ID	**0.10**‡	**0.10**	**±0.01**	**0.03**	**0.07**	**0.16**
Nguyen-6	(1 + 4)-CGP	0.32	0.42	±0.04	0.11	0.19	0.36
	(1 + 4)-CGP-ID	**0.16**‡	**0.14**	**±0.01**	**0.08**	**0.14**	**0.23**
Nguyen-7	(1 + 4)-CGP	0.45	0.30	±0.03	0.24	0.39	0.67
	(1 + 4)-CGP-ID	**0.28**‡	**0.20**	**±0.02**	**0.14**	**0.21**	**0.38**
Keijzer-6	(1 + 4)-CGP	2.52	1.61	±0.16	1.49	2.07	3.00
	(1 + 4)-CGP-ID	**1.94**‡	**1.48**	**±0.15**	**1.15**	**1.44**	**2.05**
Pagie-1	(1 + 4)-CGP	106.17	45.75	±4.57	76.43	100.59	140.56
	(1 + 4)-CGP-ID	**82.48**‡	**38.22**	**±3.82**	**55.97**	**73.78**	**97.96**
Vladislavleva-4	(1 + 4)-CGP	261.93	147.75	±14.77	144.63	163.63	466.96
	(1 + 4)-CGP-ID	**169.82**‡	**83.21**	**±8.32**	**140.13**	**144.20**	**151.26**
Korns-12	(1 + 4)-CGP	10220.62	1844.89	±184.48	8615.58	9402.35	12004.25
	(1 + 4)-CGP-ID	**9070.57**†	**1092.31**	**±109.23**	**8535.38**	**8608.195**	**9090.94**

Table 8.8: Results of the search performance evaluation for the tested symbolic regression problems evaluated by the *best-fitness-of-run* method.

Problem	Algorithm	Mean	SD	SEM	1Q	Median	3Q
		Fitness Evaluations					
Koza-1	(1 + 4)-CGP	113757	276255	±27625	1544	5084	28641
	(1 + 4)-CGP-ID	**86313**†	**63980**	**±25923**	**864**	**2408**	**15527**
Koza-2	(1 + 4)-CGP	359337	410241	±41024	13951	133352	822057
	(1 + 4)-CGP-ID	**211194**†	**313048**	**±31304**	**8127**	**46594**	**256673**
Koza-3	(1 + 4)-CGP	348474	405892	±40589	6743	109994	790608
	(1 + 4)-CGP-ID	**217992**†	**355736**	**±35573**	**3877**	**31258**	**190603**

Table 8.9: Results of the search performance evaluation for the tested symbolic regression problems Koza 1,2 and 3 evaluated by the *fitness-evaluations-to-termination* method.

Problem	Algorithm	Crossover rate [%]	Point mut. rate [%]	Insertion rate [%]	Deletion rate [%]
Parity-3	(2 + 2)-CGP	50	4	-	-
	(2 + 2)-CGP-ID	75	1	10	10
Parity-4	(2 + 2)-CGP	75	4	-	-
	(2 + 2)-CGP-ID	75	2	20	20
Parity-5	(2 + 2)-CGP	75	4	-	-
	(2 + 2)-CGP-ID	75	1	8	2
Parity-6	(2 + 2)-CGP	75	4	-	-
	(2 + 2)-CGP-ID	50	1	6	3
Parity-7	(2 + 2)-CGP	50	4	-	-
	(2 + 2)-CGP-ID	50	1	6	3
Adder 1-Bit	(2 + 2)-CGP	25	4	-	-
	(2 + 2)-CGP-ID	50	2	7.5	7.5
Adder 2-Bit	(2 + 2)-CGP	25	4	-	-
	(2 + 2)-CGP-ID	50	1	10	10
Adder 3-Bit	(2 + 2)-CGP	25	4	-	-
	(2 + 2)-CGP-ID	50	1	10	5
Multiplier 2-Bit	(2 + 2)-CGP	25	4	-	-
	(2 + 2)-CGP-ID	50	2	5	5
Multiplier 3-Bit	(2 + 2)-CGP	50	4	-	-
	(2 + 2)-CGP-ID	50	1	6	3
Demultipl. 3:8-Bit	(2 + 2)-CGP	25	4	-	-
	(2 + 2)-CGP-ID	75	2	10	10
Comparator 4x1-Bit	(2 + 2)-CGP	25	4	-	-
	(2 + 2)-CGP-ID	75	1	5	5

Table 8.10: Parametrization of the (2+2)-CGP and (2+2)-CGP-ID algorithms using subgraph crossover.

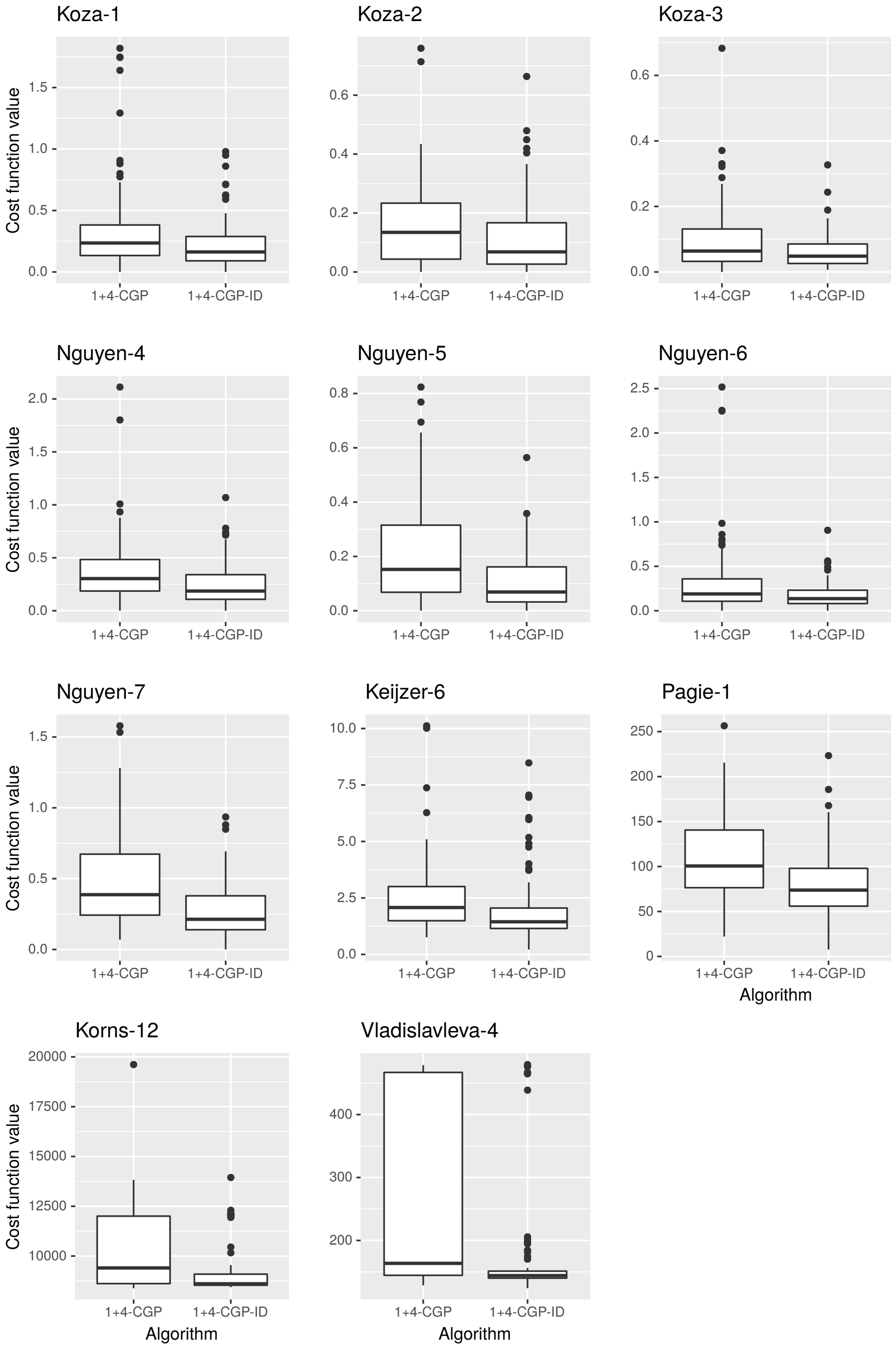


Figure 8.6: Boxplots for the results of the search performance evaluation for the tested symbolic regression problems evaluated by the *best-fitness-of-run* method.

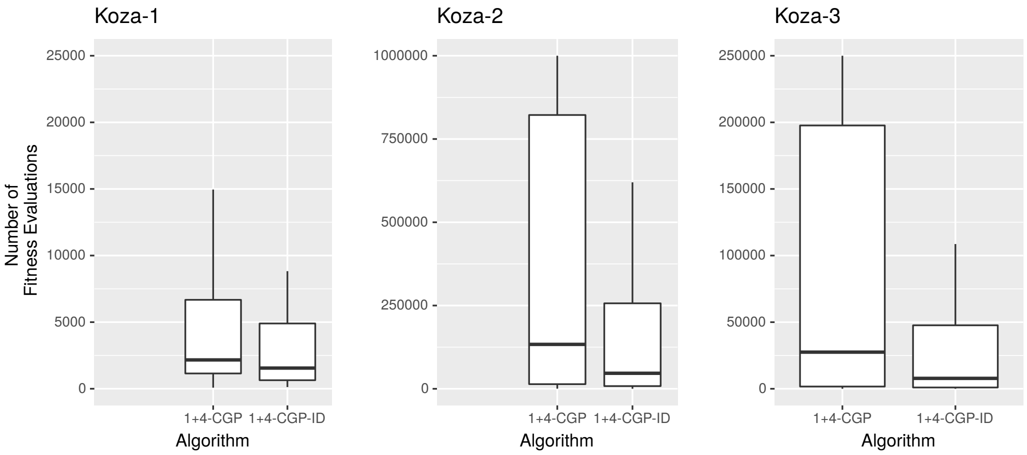


Figure 8.7: Boxplots for the results of the search performance evaluation for the problems Koza 1,2 and 3 evaluated by the *fitness-evaluations-to-termination* method.

8.5.3 Comparison to EGGP

Next, we compare three advanced CGP algorithms to a recently introduced method for evolving graphs called Evolving Graphs by Graph Programming (EGGP). EGGP has been introduced by Atkinson et al. [3]. In their experiments, Atkinson et al. compared EGGP to standard CGP and showed that EGGP performs significantly better on the majority of the tested Boolean function problems. Consequently, we chose EGGP as the baseline for our algorithm comparison. Furthermore, since we evaluated the same set of Boolean function problems as Atkinson et al. we directly compared the results of our experiments with the results in Atkinson et al. For our algorithm comparison, we chose the $(1+4)$-CGP-ID algorithm and also compared EGGP to a $(2+2)$-CGP algorithm with $\mu = 2$ and $\lambda = 2$. The $(2+2)$-CGP algorithm was equipped with the subgraph crossover technique which has described and tested in Chapter 7. Moreover, we evaluated the $(2+2)$-CGP algorithm with and without the use of the insertion and deletion technique. In the presented results, the $(2+2)$-CGP equipped with insertion and deletion mutation is denoted as $(2+2)$-CGP-ID. We evaluated important parameters like the crossover and mutation rates empirically. Moreover, we empirically tuned the parameters μ and λ and found that a configuration of $\mu = \lambda = 2$ performs best on our benchmark problems. The parameter settings for the crossover and mutation rates of the $(2+2)$-CGP and $(2+2)$-CGP-ID algorithms are shown in Table 8.10. We measured the number of fitness evaluations until a correct solution was found, similar to our search performance evaluation. To compare our results directly, we used the same evaluation method as Atkinson et al. by calculating the median value, the median absolute deviation (MAD), and the interquartile range (IQR).

Table 8.11 shows the results of the algorithm comparison for all tested Boolean

Problem	Algorithm	Median	MAD	IQR
Parity-3	(1 + 4)-CGP-ID	1928	1578	2052
	(2 + 2)-CGP	2778	2986	4564
	(2 + 2)-CGP-ID	2203	1318	2098
	EGGP	2755	1558	4836
Parity-4	(1 + 4)-CGP-ID	11920	6876	11061
	(2 + 2)-CGP	14723	12391	16432
	(2 + 2)-CGP-ID	10701	5333	8711
	EGGP	13920	5803	11629
Parity-5	(1 + 4)-CGP-ID	34622	21174	28572
	(2 + 2)-CGP	128807	83201	105579
	(2 + 2)-CGP-ID	27821	14715	25519
	EGGP	34368	15190	30054
Parity-6	(1 + 4)-CGP-ID	92466	42247	74034
	(2 + 2)-CGP	534039	505962	721456
	(2 + 2)-CGP-ID	69742	31376	46839
	EGGP	83053	33273	66611
Parity-7	(1 + 4)-CGP-ID	238426	123789	149330
	(2 + 2)-CGP	1966944	1558881	2039929
	(2 + 2)-CGP-ID	172182	72077	114928
	EGGP	197575	61405	131215
Adder 1-Bit	(1 + 4)-CGP-ID	5876	5157	6906
	(2 + 2)-CGP	8950	8951	11131
	(2 + 2)-CGP-ID	4838	3864	6377
	EGGP	5723	3020	7123
Adder 2-Bit	(1 + 4)-CGP-ID	84258	64105	85338
	(2 + 2)-CGP	191683	146445	212833
	(2 + 2)-CGP-ID	60568	40591	55450
	EGGP	74633	32863	66018
Adder 3-Bit	(1 + 4)-CGP-ID	584198	549282	640965
	(2 + 2)-CGP	2991999	2379680	3438321
	(2 + 2)-CGP-ID	378685	259886	381805
	EGGP	275180	114838	298250
Multiplier 2-Bit	(1 + 4)-CGP-ID	10196	17576	17543
	(2 + 2)-CGP	17704	20544	19383
	(2 + 2)-CGP-ID	7787	10345	10164
	EGGP	14118	5553	12955
Multiplier 3-Bit	(1 + 4)-CGP-ID	250396	236555	343552
	(2 + 2)-CGP	1024142	777862	993072
	(2 + 2)-CGP-ID	166686	118461	196298
	EGGP	1241880	437210	829223
Demultiplexer 3:8-Bit	(1 + 4)-CGP-ID	13704	6736	10797
	(2 + 2)-CGP	21047	9443	15538
	(2 + 2)-CGP-ID	9978	6394	9554
	EGGP	16763	4710	9210
Comparator 4x1-Bit	(1 + 4)-CGP-ID	272924	172932	290674
	(2 + 2)-CGP	3207723	1788937	3045088
	(2 + 2)-CGP-ID	217799	122378	182878
	EGGP	262660	84248	174185

Table 8.11: Results of the algorithm comparison.

Problem	Algorithm	Mean Active Function Node Range	SD	SEM	1Q	Median	3Q
Parity-3	$(1+4)$-CGP	33.33	4.54	±0.45	31.00	33.00	35.00
	$(1+4)$-CGP-ID	38.31	7.82	±0.78	34.00	39.00	43.00
Parity-4	$(1+4)$-CGP	36.18	3.97	±0.39	33.75	36.00	39.00
	$(1+4)$-CGP-ID	50.80	6.04	±0.60	47.00	51.00	55.00
Parity-5	$(1+4)$-CGP	35.02	4.13	±0.41	32.75	34.75	37.00
	$(1+4)$-CGP-ID	58.81	5.17	±0.52	56.00	59.00	62.00
Parity-6	$(1+4)$-CGP	35.18	4.52	±0.45	32.00	34.00	37.25
	$(1+4)$-CGP-ID	52.21	6.73	±0.63	47.00	53.50	57.00
Parity-7	$(1+4)$-CGP	35.21	4.27	±0.43	33.00	35.00	38.00
	$(1+4)$-CGP-ID	51.83	6.80	±0.68	48.00	51.00	56.25
Adder 1-Bit	$(1+4)$-CGP	38.73	5.67	±0.56	35.00	38.00	42.00
	$(1+4)$-CGP-ID	38.98	5.17	±0.52	36.00	39.00	42.25
Adder 2-Bit	$(1+4)$-CGP	36.00	4.04	±0.40	33.00	36.00	38.25
	$(1+4)$-CGP-ID	52.47	9.37	±0.94	47.00	52.00	60.00
Adder 3-Bit	$(1+4)$-CGP	36.82	4.13	±0.41	34.00	36.00	39.00
	$(1+4)$-CGP-ID	45.01	6.59	±0.66	40.00	44.00	49.00
Multiplier 2-Bit	$(1+4)$-CGP	36.31	3.81	±0.38	34.00	36.00	39.00
	$(1+4)$-CGP-ID	38.00	4.95	±0.49	35.00	37.00	41.00
Multiplier 3-Bit	$(1+4)$-CGP	36.46	4.74	±0.47	33.00	36.00	39.00
	$(1+4)$-CGP-ID	41.28	5.47	±0.55	37.00	40.50	46.00
Demultiplexer 3:8-Bit	$(1+4)$-CGP	35.53	4.16	±0.42	32.75	35.00	38.00
	$(1+4)$-CGP-ID	42.69	6.00	±0.60	39.00	42.00	46.25
Comparator 4x1-Bit	$(1+4)$-CGP	30.38	3.53	±0.35	28.00	30.00	33.00
	$(1+4)$-CGP-ID	32.60	5.40	±0.54	29.00	32.00	36.00

Table 8.12: Results of the active function node range analysis.

function problems. It is visible that the median values of the $(2+2)$-CGP-ID and EGGP are on the same level. Moreover, it is also evident that we achieved a lower median value of fitness evaluations for the $(2+2)$-CGP-ID algorithm on all problems except the Adder 3-Bit problem of the tested problems. Please note that the results for EGGP have been directly taken from the work of Atkinson et al.

8.5.4 Active Function Node Range Analysis

To measure the exploration with and without our proposed mutation in phenotype space, we analyzed the range of the active function nodes. We measured the number of active function nodes of the best individual in each generation and calculated the range at the end of each run.

We performed 100 runs for each algorithm and allowed a budget of 2500 generations. Afterward, we completed the statistical evaluation on the range values for the $(1+4)$-CGP and $(1+4)$-CGP-ID algorithm. To visualize the behavior of the active function node range, we calculated the smoothed conditional mean over a predefined budget of generations for each benchmark problem. As smoothing method we used the generalized additive model approach (GAM) [38, 39].

Table 8.12 and Table 8.13 show the results of the function node range analysis for all tested boolean function and symbolic regression problems. It is clearly visible that

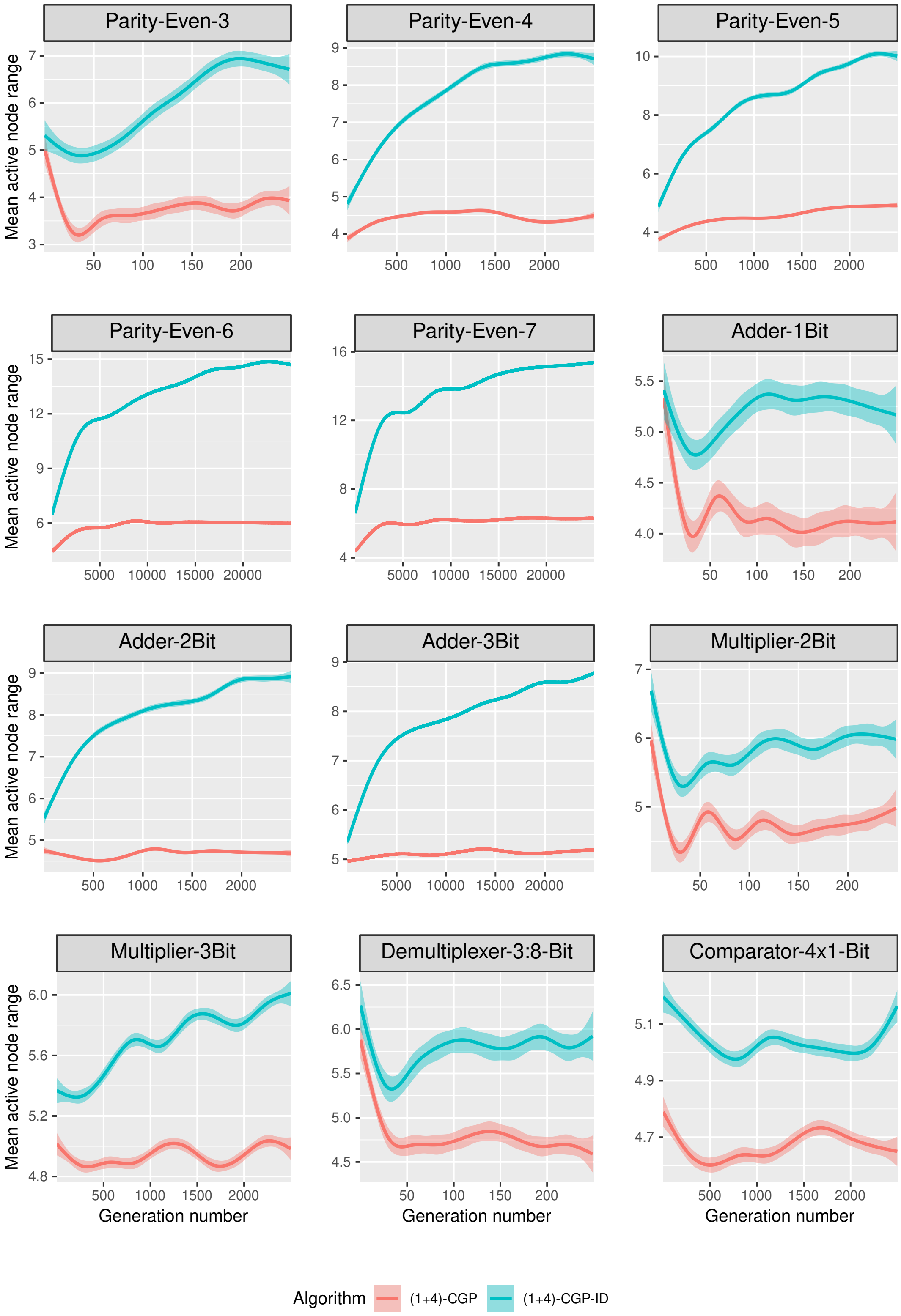


Figure 8.8: Visualization of the smoothed conditional mean for the active function node range analysis over a certain number of generations on all tested Boolean function problems.

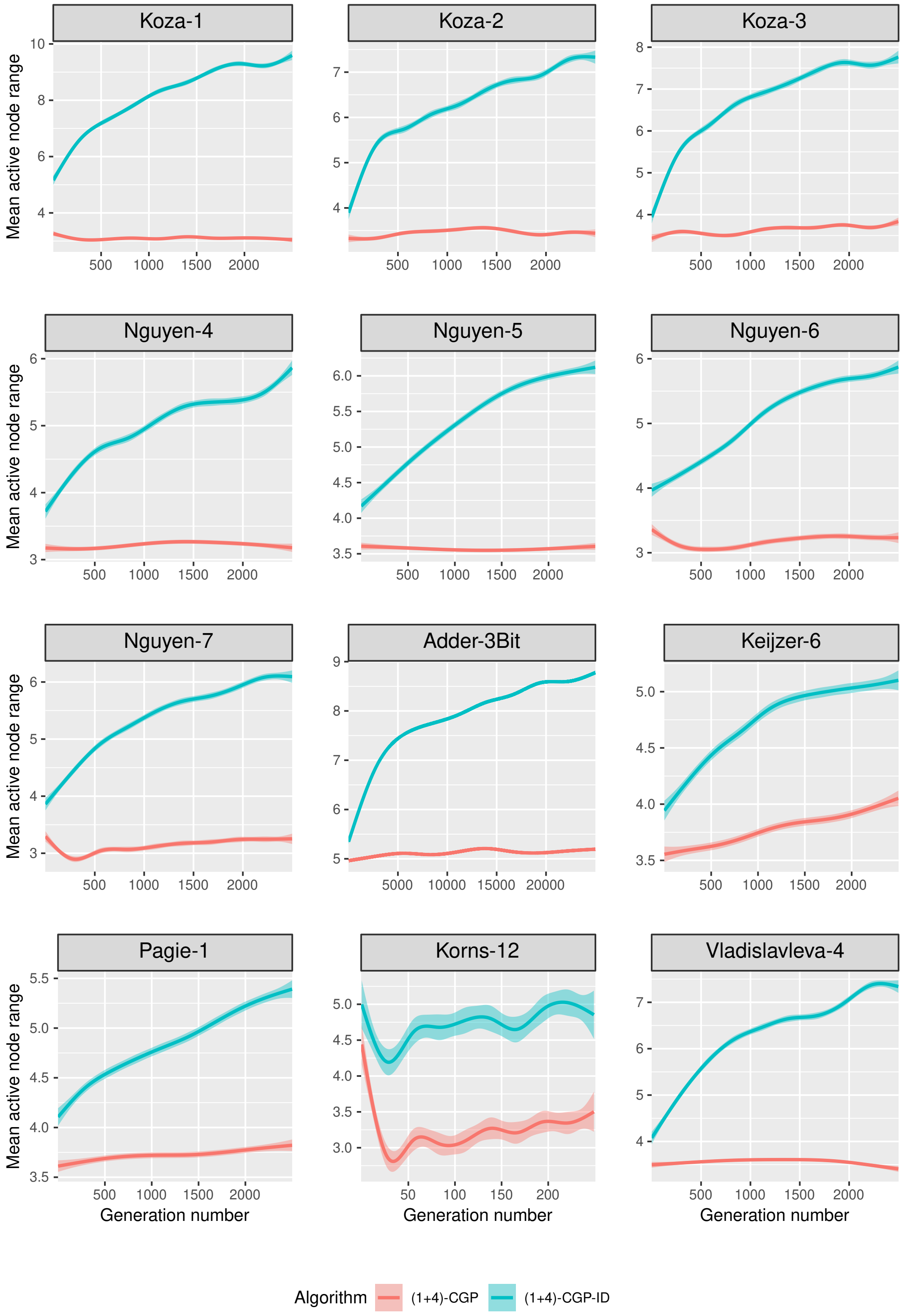


Figure 8.9: Visualization of the smoothed conditional mean for the active function node range analysis over a certain number of generations on all tested symbolic regression problems.

Problem	Algorithm	Mean Active Function Node Range	SD	SEM	1Q	Median	3Q
Koza-1	(1 + 4)-CGP	8.63	3.36	±0.33	6.75	8.00	11.00
	(1 + 4)-CGP-ID	14.16	4.93	±0.49	10.00	14.00	17.25
Koza-2	(1 + 4)-CGP	10.33	4.20	±0.42	8.00	10.00	13.00
	(1 + 4)-CGP-ID	15.10	4.83	±0.48	11.75	15.00	17.50
Koza-3	(1 + 4)-CGP	11.04	3.72	±0.37	8.75	11.00	13.00
	(1 + 4)-CGP-ID	15.93	4.87	±0.48	13.00	16.00	19.25
Nguyen-4	(1 + 4)-CGP	9.51	3.62	±0.36	7.00	10.00	12.00
	(1 + 4)-CGP-ID	14.79	4.53	±0.45	12.00	15.00	18.00
Nguyen-5	(1 + 4)-CGP	10.31	4.49	±0.45	6.75	9.00	13.00
	(1 + 4)-CGP-ID	14.75	4.33	±0.43	12.00	15.00	17.00
Nguyen-6	(1 + 4)-CGP	9.27	4.02	±0.40	6.00	9.00	11.00
	(1 + 4)-CGP-ID	13.88	4.82	±0.48	10.00	14.00	17.25
Nguyen-7	(1 + 4)-CGP	8.52	3.97	±0.39	5.75	8.00	11.00
	(1 + 4)-CGP-ID	13.9	4.70	±0.47	10.75	14.00	17.00
Keijzer-6	(1 + 4)-CGP	11.61	4.75	±0.47	8.00	11.00	14.25
	(1 + 4)-CGP-ID	13.38	4.73	±0.47	10.00	13.00	16.00
Pagie-1	(1 + 4)-CGP	12.05	5.37	±0.53	8.00	12.00	15.00
	(1 + 4)-CGP-ID	16.83	7.35	±0.73	11.00	15.00	22.00
Vladislavleva-4	(1 + 4)-CGP	16.02	5.55	±0.55	13.00	16.00	20.00
	(1 + 4)-CGP-ID	20.93	7.04	±0.70	17.00	21.00	25.00
Korns-12	(1 + 4)-CGP	7.14	3.25	±0.32	5.00	7.00	9.00
	(1 + 4)-CGP-ID	9.54	4.39	±0.44	6.00	9.00	12.25

Table 8.13: Results of the active function node range analysis.

the range of active function nodes of the (1 + 4)-CGP-ID is greater compared to the (1 + 4)-CGP for the majority of our tested problems. Figure 8.8 and Figure 8.9 illustrate the smoothed conditional mean of the active function node range over a certain number of generations.

8.6 Discussion

The primary concern of our experiments was to find significant contributions of the insertion and deletion mutation technique to the search performance of CGP. The results of our experiments showed beneficial effects on a diverse set of symbolic regression and Boolean function problems. One point which should be discussed is the runtime measurement of our experiments. On the one hand, we observed a reduced amount of fitness evaluations when the insertion and deletion mutation techniques were in use for all tested problems. Our runtime measurement revealed that the beneficial effects were only significant when the complexity of the problem is high or when an expensive fitness function is used. Moreover, the use of the insertion and deletion mutation techniques needs a certain amount of computational time. However, it is visible that the use of our proposed mutations showed good runtime results on the more complex Boolean function problems such as the Parity-7, Adder 3-Bit, and Multiplier 3-Bit problems. We must report that we performed

our experiments with a naive Java implementation of both mutation techniques. A more efficient implementation is certainly possible.
Our experiments also addressed the question in which way the insertion and deletion mutation techniques improve the search performance of CGP. The analytic part of our experiments was devoted to a range analysis of the active function node of the best individual in the population. The results of this experiment indicate that our proposed mutations can lead to more exploration of the phenotype space. Another answer and explanation to the question of the effectiveness of our proposed mutations may be found in the work of Goldman and Punch. In their work, Goldman and Punch [34] analyzed the evolutionary mechanisms of traditional CGP and concluded that

> ”We found large sections of the genome were never used by any ancestor of the final solution. Furthermore, offspring almost never include active nodes that were inactive in their direct parent but active in a previous ancestor.”
>
> Goldman and Punch [34, p. 359]

. Moreover, regarding the actual behavior of CGP, Goldman, and Punch also concluded that

> ”CGP genomes include a surprising amount of redundant and unused nodes.”
>
> Goldman and Punch [34, p. 372]

Based on the very detailed and precise analysis of Goldman and Punch, we assume that our proposed mutations cause more activity in these large and normally unused sections of the CGP genome by activating inactive function nodes. Another hypothetical explanation is based on the outcome of another work by Goldman and Punch [32]. For an investigation of the length bias and search limitations in CGP the Goldman and Punch found that CGP has an innate parsimony pressure, which makes it very difficult to evolve individuals with a high percentage of active nodes. Based on this finding, we assume that the use of the insertion and deletion mutation techniques counteract the observed innate parsimony pressure. One indicator for this assumption is the outcome of our active function node analysis. For the $(1+4)$-CGP-ID algorithm, we observed a wider range of active function nodes which have been processed throughout the performed evolutionary runs.

Our comparison with EGGP showed that the use of our proposed mutations in combination with the subgraph crossover indicates that these advanced techniques are beneficial for the use of CGP. Furthermore, on some of our tested problems, we achieved a lower median value for the $(2+2)$-CGP-ID algorithm when compared to EGGP. However, for more significant and meaningful statements about the current state of EGGP and CGP, a more comprehensive study is needed and should include different problem domains. For the field of graph-based Genetic Programming, this point is of high importance because there is comparatively only a little knowledge about the search performance of CGP and EGGP in other problem domains. Moreover, EGGP and CGP have been mostly evaluated with Boolean function problems in the past, which resulted in a one-sided state of knowledge. Therefore, we think that comprehensive comparative studies are needed to expand the current state of knowledge.

Addressing the reasons of the effectiveness of the $(2+2)$-CGP-ID algorithm, we have to acknowledge that we do not have any results and answers to the question in which way the combination of subgraph crossover and our proposed mutations contribute to the search performance of CGP. The results of our experiments open two issues that have to be tackled with our future work: In the first place, we have to find answers in which way the $(2+2)$-CGP-ID algorithm contributes to the search performance of CGP. To achieve insight into the detailed functional mechanism of the $(2+2)$-CGP and $(2+2)$-CGP-ID algorithm, we have to understand the proposed methods in detail. As a first step forward, we think a separate investigation of exploitation and exploration effects of the $(2+2)$-CGP and $(2+2)$-CGP-ID algorithm would be helpful. We also have to tackle the question of why small population sizes are generally successful in the Boolean domain. Since the effectiveness of the $(1+4)$-CGP in the Boolean domain is well known in the field of CGP [97, 99], our experiments with the $(2+2)$-CGP-ID algorithm underline the effectiveness of small population sizes in the Boolean problem domain. Consequently, there is a need for more insight into the observed conditions of our experiments.

8.7 Conclusion

Within this chapter, we proposed two new phenotypic mutation techniques and took a step toward advanced phenotypic mutations in CGP. The results of our experiments clearly show that our proposed methods can be beneficial for the use of CGP. Our experiments also clearly show that the insertion and deletion mutation techniques can significantly improve the search performance of CGP. We also compared CGP to another *state-of-the-art* method for evolving graphs and showed that advanced crossover and mutation methods allow CGP to perform well.The analytic part of our experiments indicate that our proposed mutations enable a wider search in high fitness regions within the search space. We also compared the outcome of the active node range analysis to former studies and provided hypothetical assumptions in which way weaknesses and limitations of the standard $(1+\lambda)$-CGP with point mutation are counteracted by our proposed mutations. However, we would like to stress that these assumptions are of hypothetical nature and have to be analyzed by appropriate experiments.

Summary of the results of the experiments:

- Phenotypic mutations can be used to improve the search performance of mutation-only CGP
- The proposed mutations can be used in combination with phenotypic recombination to improve the search performance
- Analytic experiments indicate that the use of the proposed mutations enables a wider search in phenotype space

9 Evaluation and Analysis

9.1 Introduction

The primary purpose of this chapter is to present the result of a final evaluation of the methods proposed in this thesis. To classify the significance of the presented methods and the respective algorithmic setup, the results are compared to the traditional $(1+4)$-CGP algorithm. Therefore, this evaluation study intends to investigate if the use of the presented techniques can be considered as more effective than the use of the $(1+4)$-CGP algorithm. We will compare different evolutionary algorithms. Some of them are consequently equipped with the methods, which have been proposed in the previous chapters. A list of the algorithms which will be compared in this study is given in Table 9.1. Algorithms which are equipped with techniques presented in this thesis are marked with a $\star$. The evaluation part of this chapter is covered in Section 9.2.

Another part of this chapter is devoted to an analysis of certain algorithms compared in the evaluation sections. On one hand, this analysis includes an investigation between different settings of the $(1+\lambda)$-CGP and CGP when used with a canonical evolutionary algorithm. This part of the analysis is presented in Section 9.3. Another part of the analysis in devoted to an investigation of the effects caused by the use of the subgraph crossover. The analysis and the corresponding experiments are presented in Section 9.3.

At the end of this chapter, we present the results of redundancy and fitness space analysis in Section 9.5. This analysis sheds more light on the correlation of the level of redundancy and the size of the space of fitness values. In this way, the analysis intends to clarify resulting question on the parametrization of the length of the CGP genotype.

9.1.1 The Need for a New Comparative Study

The results of the meta-evolution in Chapter 7 indicate that the tuning of crossover algorithms is a complex task and makes fair comparison difficult. This is because crossover-based CGP algorithms require the configuration of more parameters as the $(1+\lambda)$-CGP. The outcome of the parameter tuning in Chapter 7 also indicates that merely relying on the results of the meta-evolution can lead to ineffective parameter settings and unfair comparisons. Since the meta-evolution is performed using an evolutionary algorithm, it is possible that the meta-evolution evolves toward a local optimum, which results in ineffective parameter settings. Therefore, a solid and precise parameter tuning needs additional steps, which will be described in the

Table 9.1: List of the CGP algorithms.

List CGP algorithms

$(1+4)$-CGP	Standard $(1+4)$-CGP algorithm
$(1+\lambda)$-CGP	Standard $(1+\lambda)$-CGP algorithm
$(1+\lambda)$-CGP-ID $\star$	$(1+\lambda)$-CGP algorithm with insertion and deletion mutation
$(\mu+\lambda)$-CGP (Subgraph) $\star$	$(\mu+\lambda)$-CGP algorithm with subgraph crossover
$(\mu+\lambda)$-CGP (Block) $\star$	$(\mu+\lambda)$-CGP algorithm with block crossover
$(\mu+\lambda)$-CGP-ID (Subgraph) $\star$	$(\mu+\lambda)$-CGP algorithm with insertion and deletion mutation and subgraph crossover
$(\mu+\lambda)$-CGP-ID (Block) $\star$	$(\mu+\lambda)$-CGP algorithm with insertion and deletion mutation and block crossover
Canonical-CGP (Subgraph) $\star$	Standard canonical EA with subgraph crossover
Canonical-CGP (Block) $\star$	Standard canonical EA with block crossover
Real-valued CGP	Real-valued CGP with decimal representation
Adaptive real-valued CGP $\star$	Real-valued CGP algorithm with self-adaptive strategy

following subsection.

Another reason for a new comparative study is that a comparison with a bigger set of benchmark problems is needed to shed more light on the role of crossover and advanced mutations in CGP. For instance, in Chapter 7 only 8 problems were included in a comparative study.

The last reason for this evaluation study is the fact that more detailed research on the question of crossover itself is needed. The study in Chapter 7 answers the question that a crossover-based algorithm can be more effective than the standard $(1+\lambda)$ algorithm, but a comprehensive and detailed study on this topic is still missing.

9.1.2 Parameter Tuning

To determine efficient parameter settings for fair comparisons, we use an approach to parameter tuning for CGP, which has been described and used in Chapter 5 and Chapter 7. The parameter tuning is done in three steps. In the first place, a set of well-performing parameter settings is determined by a meta evolutionary algorithm. This determined set of parameters is validated manually, and the best configuration is chosen for further tuning. Finally, the best performing configuration is fine-tuned on an empirical level.

9.2 Evaluation of the proposed methods

9.2.1 Experimental Setup

The experimental setup of our comparative study is very similar to the setup in Chapter 7. We performed experiments on the same problems as in Chapter 7 in the problem domains of symbolic regression, Boolean functions, and image operator design. We also included more complex Boolean function problems. To evaluate the search performance of the tested algorithms, we measured the number of fitness evaluations until the CGP algorithm terminated successfully (*fitness-evaluations-to-success*) and the best fitness value, which was found after a predefined number of generations (*best-fitness-of-run*). In addition to the mean values of the measurements, we calculated the standard deviation (SD) and the standard error of the mean (SEM). To classify the significance of our results, we used the Mann-Whitney-U-Test. The mean values are denoted $a^\dagger$ if the p-value is less than the significance level 0.05 and $a^\ddagger$ if the p-value is less than the significance level 0.01 compared to the $(1+4)$-CGP. Note that the mean values are **only** denoted with the significance level marker if the result of a certain algorithm is better than the result of the $(1 + 4)$-CGP. We performed 100 independent runs with different random seeds except the complex and computing-intensive Even-Parity 8 problem for which we performed only 30 runs. The elitism size was set to 1. For the use of the insertion and deletion mutation, we chose a minimum of 2 active function nodes and maximum of 100 active function nodes. The levels back parameter l was set to ∞. For the block crossover we used a block size of 2 function nodes.

9.2.2 Benchmarks

Symbolic Regression

We chose nine symbolic regression problems from the work of Clegg et al. [13] and McDermott et al. [90]. The functions of the problems are shown in Table 9.2. A training data set U$[a, b, c]$ refers to c uniform random samples drawn from a to b inclusive and E$[a, b, c]$ refers to a grid of points evenly spaced with an interval of c, from a to b inclusive. The Koza function set consisted of eight mathematical functions ($+$, $-$, $*$, $/$, sin, cos, $\ln(|n|)$, e^n) and the Keijzer function set of five mathematical functions ($+$, $*$, $\frac{1}{n}$, $-n$,$\sqrt{n}$). The fitness of the individuals was represented by a cost function value. The cost function was defined by the sum of the absolute difference between the real function values and the values of an evaluated individual.

We evaluated the simpler symbolic regression problems Koza 1,2 & 3 with the *fitness-evaluation-to-termination* method. The other, more complex, symbolic regression problems were evaluated with the *best-fitness-of-run* method. We defined a maximum number of $8 \cdot 10^7$ fitness evaluations for these three experiments. The reason for our choice of these three problems is the fact that we can find an ideal solution more

Table 9.2: List of symbolic regression benchmarks.

Problem	**Objective Function**	**Vars**	**Training Set**	**Function Set**
Koza-1	$x^4 + x^3 + x^2 + x$	1	U[-1,1,20]	Koza
Koza-2	$x^5 - 2x^3 + x$	1	U[-1,1,20]	Koza
Koza-3	$x^6 - 2x^4 + x^2$	1	U[-1,1,20]	Koza
Nguyen-4	$x^6 + x^5 + x^4 + x^3 + x^2 + x$	1	U[-1,1,20]	Koza
Nguyen-5	$\sin(x^2)\cos(x) - 1$	1	U[-1,1,20]	Koza
Nguyen-6	$\sin(x) + \sin(x + x^2)$	1	U[-1,1,20]	Koza
Nguyen-7	$\ln(x + 1) + \ln(x^2 + 1)$	1	U[0,2,20]	Koza
Keijzer-6	$\sum_i^x 1/i$	1	E[1,50,1]	Keijzer
Pagie-1	$1/(1 + x^{-4}) + 1/(1 + y^{-4})$	2	E[-5,5,0.4]	Koza

Table 9.3: Function sets for the Boolean function problems.

Problems	**Function Set**
Parity-Even 3,4,5,6,7,8	AND, OR, NAND, NOT, NOR
Adder-1,2,3, Subtractor-2, Multiplier-2,3 Bit	AND, OR, NAND, NOR

likely on average than the other more complex benchmark problems, which require a huge number of fitness evaluations to find an ideal solution. For the more complex problems, we measured the best fitness after a budget of 10^4 fitness evaluations.

Boolean functions

In the Boolean domain, we chose six Parity-Even problems with $n = 3$ to 8 Boolean inputs. The goal was to find a program that produces the value of the Boolean even parity depending on the n independent inputs. The fitness was represented by the number of fitness cases for which the candidate solution failed to generate the correct value of the Even-Parity function.

We also investigated multiple output problems as the 1,2 & 3¸-Bit-Adder, 2-Bit-Subtractor, 2-Bit- and 3-Bit-Multiplier. To evaluate the fitness of the individuals on the multiple output problems, we defined the fitness value of an individual as the number of different bits to the corresponding truth table. When this number became zero, the algorithm successfully terminated. We evaluated the problems with the *fitness-evaluations-to-success* method. The function sets for the experiments with the Parity-Evon problems and the multiple output problems are shown in Table 9.3.

Image operator design

We chose the Gaussian noise and Salt & Pepper noise reduction problems, which are described Section 7. The fitness function was defined as the *Mean Difference per Pixel*. The image data consisted of the Lena test image set with an image size of 128x128 pixel , which includes the input images and reference image is shown in Figure 9.2,9.1 and 9.3. We evaluated all image operator design problems with the *fitness-evaluations-to-success* method. The function set for the image operator

Table 9.4: Function set for the image operator design problems.

Function	**Description**
OR	Bitwise OR
OR*	Bitwise OR with one inverted input
AND	Bitwise AND
NAND	Bitwise NAND
XOR	Bitwise XOR
ADD	Addition
ADD*	Addition with saturation
MAX	Maximum
MIN	Minimum
AVG	Average
ID	Indenity
INV	Inversion
SWAP	Swap upper and lower nibble
RIGHT SHIFT ONE	Right bit-shift by one
RIGHT SHIFT TWO	Right bit-shift by two
CONSTANT	Return of value 255

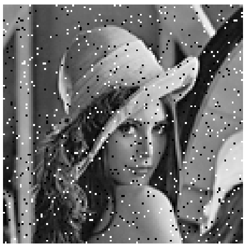


Figure 9.1: Salt & Pepper noise

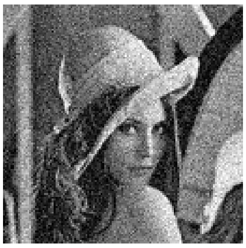


Figure 9.2: Gaussian noise

Figure 9.3: Original image

design problems is shown in Table 9.4.

All benchmark problems which are used for this study are listed in Table 9.5.

9.2.3 Meta-optimization

Similar to Chapter 7, we performed meta-optimization experiments with the intention to compare CGP algorithms in a fair way. Moreover, we tuned significant parameters for all used CGP algorithms on the set of benchmark problems. We used the meta-optimization extension package of the Java Evolutionary Computation Research System (ECJ). The parameters for the respective CGP algorithms are given in Table 9.6. For the meta-level, we used a canonical GA. The setting of the meta-level GA is shown in Table 9.7. The ranges for the number of nodes are oriented with the parameter settings found in Chapter 5. Meta-evolution is very costly in terms of computational effort, which is necessary to find an optimal parameter setting. Furthermore, since GP benchmark problems can be very noisy in terms of

Table 9.5: List of all benchmark problems for the comparison.

Problem Domain	**Problems**
Symbolic Regression	Koza-1,2,3; Nguyen-4,5,6,7; Keijzer-6; Pagie-1
Boolean Functions	Parity-Even 3,4,5,6,7,8; Adder-1,2,3 Subtractor-2, Multiplier-2,3 Bit
Image Operator Design	Gaussian noise reduction, Salt & Pepper noise reduction

Table 9.6: Parameter space explored by meta evolution for the fundamental CGP algorithms.

$(1+4)$-CGP

Parameter	**Description**	**Range**
N	number of nodes	[10,4000]
M_p	point mutation rate[%]	[1,20]

$(1+\lambda)$-CGP

λ	number of offspring	[1,1000]
N	number of nodes	[10,4000]
M_p	point mutation rate[%]	[1,20]

$(1+\lambda)$-CGP-ID

λ	number of offspring	[1,1000]
N	number of nodes	[10,4000]
M_p	point mutation rate[%]	[1,20]
M_i	insertion rate[%]	[1,20]
M_d	deletion rate[%]	[1,20]

Canonical-CGP

N	number of nodes	[10,4000]
M_p	point mutation rate[%]	[1,20]
C	crossover rate[%]	[10,100]
P	population size	[5,1000]
T	tournament size	[2,20]

$(\mu+\lambda)$-CGP

μ	number of parents	[2,150]
λ	number of offspring	[1,1000]
N	number of nodes	[10,4000]
M_p	point mutation rate[%]	[1,20]
C	crossover rate[%]	[10,100]

$(\mu+\lambda)$-CGP-ID

μ	number of parents	[2,150]
λ	number of offspring	[1,1000]
N	number of nodes	[10,4000]
M_p	point mutation rate[%]	[1,20]
M_i	insertion rate[%]	[1,20]
M_d	deletion rate[%]	[1,20]
C	crossover rate[%]	[10,100]

Table 9.7: Configuration of the meta-level GA.

Property	**Setting**
Maximum generations	200
Population size	50
Mutation rate	$1/n$
Mutation tape	gaussian mutation
Tournament selection size	4
Crossover rate	0.7
Crossover type	intermediate recombination
Evaluation method	best-fitness-of-run
Number of trials	4

finding the ideal solution, we used a common approach of fine-tuning the parameters which have also been used for the work presented in Chapter 5. The meta-evolution is repeated several times, and the best settings are used to find the ideal algorithm setting. This is done afterward with manual fine-tuning of the respective parameters.

Tables 9.8, 9.9, 9.10, and 9.11 show the results of the meta-optimization. The results in the Boolean domain reveal an effective parametrization pattern, characterized by the choice of a very small population size and an extremely high number of function nodes. This finding also holds for the crossover-based algorithms. In the symbolic regression domain, the number of function nodes is, in average smaller compared to the Boolean function domain. The results indicate that bigger population sizes seem to work more effectively in this problem domain. For the image operator design problems, very small population sizes and an extremely high number of function nodes seem to be a good choice which is similar to the findings in the Boolean domain. It can also be observed that the crossover rate for the symbolic regression problems is higher in average when compared to the rates determined in the Boolean domain.

9.2.4 Experiments

For the algorithm comparison, which we performed on the basis of the results of our experiments, we chose the traditional $(1 + 4)$-CGP as the baseline algorithm. We evaluated all problems with the respective parameter setting, which had been determined in meta-optimization part of this study.

The results of our experiments in the symbolic regression domain are shown in Table 9.12 and Table 9.13. It is visible that the Canonical-CGP algorithm with subgraph crossover performs better than the $(1 + 4)$-CGP on all tested problems. Moreover, on some of the tested problems the use of the block crossover and the proposed advanced mutations leads to better results when compared to the $(1 + 4)$-CGP.

Table 9.14 and Table 9.15 show the results of the algorithm comparison in the Boolean domain. It is clearly visible that the choice of a $\lambda = 1$ results in a reduced

Table 9.8: Results of the meta optimization for the single output Boolean function problems.

Problem	**Algorithm**	N	M_p	M_i	M_d	C	μ	λ	P	T
Parity-3	$(1+4)$-CGP	2000	1	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	2000	1	–	–	–	–	1	–	–
	$(1+\lambda)$-CGP-ID	2000	1	5.0	5.0	–	–	1	–	–
	Canonical-CGP (Subgraph)	2000	1	–	–	70	–	–	10	2
	Canonical-CGP (Block)	2000	1	–	–	50	–	–	5	2
	$(\mu+\lambda)$-CGP (Subgraph)	2000	1	–	–	90	4	1	–	–
	$(\mu+\lambda)$-CGP (Block)	2000	1	–	–	50	4	1	–	–
	$(\mu+\lambda)$-CGP-ID (Block)	2000	1	5.0	5.0	50	4	1	–	–
	Real-valued-CGP	4000	1	–	–	70	–	–	5	2
	Adaptive Real-valued-CGP	4000	[1,20]	–	–	[10,90]	–	–	5	2
Parity-4	$(1+4)$-CGP	1500	1	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	1500	1	–	–	–	–	1	–	–
	$(1+\lambda)$-CGP-ID	1500	1	5.0	5.0	–	–	1	–	–
	Canonical-CGP (Subgraph)	1500	1	–	–	70	–	–	5	2
	Canonical-CGP (Block)	1500	1	–	–	50	–	–	5	2
	$(\mu+\lambda)$-CGP (Subgraph)	1500	1	–	–	90	4	1	–	–
	$(\mu+\lambda)$-CGP (Block)	1500	1	–	–	70	4	1	–	–
	$(\mu+\lambda)$-CGP - ID (Subgraph)	1500	1	5.0	5.0	90	4	1	–	–
	Real-valued-CGP	3000	1	–	–	70	–	–	5	2
	Adaptive Real-valued-CGP	3000	[1,20]	–	–	[10,90]	–	–	5	2
Parity-5	$(1+4)$-CGP	2000	1	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	2000	1	–	–	–	–	1	–	–
	$(1+\lambda)$-CGP-ID	2000	1	5.0	5.0	–	–	1	–	–
	Canonical-CGP (Subgraph)	1000	1	–	–	70	–	–	5	2
	Canonical-CGP (Block)	2000	1	–	–	20	–	–	5	2
	$(\mu+\lambda)$-CGP (Subgraph)	2000	1	–	–	25	4	1	–	–
	$(\mu+\lambda)$-CGP (Block)	2000	1	–	–	25	4	1	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	2000	1	5.0	5.0	25	4	1	–	–
	Real-valued-CGP	3000	1	–	–	70	–	–	5	2
	Adaptive Real-valued-CGP	3000	[1,10]	–	–	[10,90]	–	–	5	2
Parity-6	$(1+4)$-CGP	2000	1	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	2000	1	–	–	–	–	1	–	–
	$(1+\lambda)$-CGP-ID	2000	1	5.0	5.0	–	–	1	–	–
	Canonical-CGP (Subgraph)	2000	1	–	–	70	–	–	5	2
	Canonical-CGP (Block)	2000	1	–	–	70	–	–	5	2
	$(\mu+\lambda)$-CGP (Subgraph)	2000	1	–	–	90	4	1	–	–
	$(\mu+\lambda)$-CGP (Block)	2000	1	–	–	20	4	1	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	2000	1	5.0	5.0	50	4	1	–	–
	Real-valued-CGP	3000	1	–	–	70	–	–	5	2
	Adaptive Real-valued-CGP	3000	[1,10]	–	–	[10,90]	–	–	5	2
Parity-7	$(1+4)$-CGP	2500	1	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	2500	1	1.0	1.0	–	–	1	–	–
	$(1+\lambda)$-CGP-ID	2500	1	–	–	–	–	–	–	–
	Canonical-CGP (Subgraph)	2500	1	–	–	50	–	–	5	2
	Canonical-CGP (Block)	2500	1	–	–	50	–	–	5	2
	$(\mu+\lambda)$-CGP (Subgraph)	2500	1	–	–	50	4	1	–	–
	$(\mu+\lambda)$-CGP (Block)	2500	1	–	–	50	4	1	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	2500	1	2.5	2.5	50	4	1	–	–
	Real-valued-CGP	3000	1	–	–	70	–	–	5	2
	Adaptive Real-valued-CGP	3000	[1,10]	–	–	[10,90]	–	–	5	2
Parity-8	$(1+4)$-CGP	2000	1	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	2000	1	–	–	–	–	1	–	–
	$(1+\lambda)$-CGP-ID	2000	1	1.0	1.0	–	–	1	–	–
	Canonical-CGP (Subgraph)	2000	1	–	–	50	–	–	5	2
	Canonical-CGP (Block)	2000	1	–	–	50	–	–	5	2
	$(\mu+\lambda)$-CGP (Subgraph)	2000	1	–	–	50	4	1	–	–
	$(\mu+\lambda)$-CGP (Block)	2000	1	–	–	50	4	1	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	2000	1	2.5	2.5	50	4	1	–	–
	Real-valued-CGP	2000	1	–	–	70	–	–	5	2
	Adaptive Real-valued-CGP	2000	[1,10]	–	–	[10,90]	–	–	5	2

Table 9.9: Results of the meta optimization for the multiple output Boolean function problems.

Problem	**Algorithm**	N	M_p	M_i	M_d	C	μ	λ	P	T
Adder-1Bit	$(1+4)$-CGP	150	3	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	150	3	–	–	–	–	1	–	–
	$(1+\lambda)$-CGP-ID	150	3	5.0	5.0	–	–	1	–	–
	Canonical-CGP (Subgraph)	150	3	–	–	50	–	–	5	2
	Canonical-CGP (Block)	150	3	–	–	20	–	–	5	2
	$(\mu+\lambda)$-CGP (Subgraph)	150	3	–	–	25	4	1	–	–
	$(\mu+\lambda)$-CGP (Block)	150	3	–	–	25	4	1	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	150	2	10.0	10.0	50	4	1	–	–
	Real-valued-CGP	150	3	–	–	20	–	–	5	2
	Adaptive Real-valued-CGP	150	[1,10]	–	–	[10,50]	–	–	5	2
Adder-2Bit	$(1+4)$-CGP	100	3	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	150	1	–	–	–	–	1	–	–
	$(1+\lambda)$-CGP-ID	150	1	10.0	5.0	–	–	1	–	–
	Canonical-CGP (Subgraph)	150	1	–	–	20	–	–	5	2
	Canonical-CGP (Block)	150	1	–	–	20	–	–	5	2
	$(\mu+\lambda)$-CGP (Subgraph)	150	1	–	–	25	4	1	–	–
	$(\mu+\lambda)$-CGP (Block)	150	1	–	–	25	4	1	–	–
	$(\mu+\lambda)$-CGP (Subgraph)	150	1	10.0	10.0	50	16	1	–	–
	Real-valued-CGP	150	1	–	–	50	–	–	5	2
	Adaptive Real-valued-CGP	150	[1,10]	–	–	[10,50]	–	–	5	2
Adder-3Bit	$(1+4)$-CGP	2000	1	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	2000	1	–	–	–	–	1	–	–
	$(1+\lambda)$-CGP-ID	2000	1	2.5	2.5	–	–	1	–	–
	Canonical-CGP (Subgraph)	2000	1	–	–	50	–	–	5	2
	Canonical-CGP (Block)	2000	1	–	–	50	–		5	2
	$(\mu+\lambda)$-CGP (Subgraph)	2000	1	–	–	50	8	1	–	–
	$(\mu+\lambda)$-CGP (Block)	2000	1	–	–	25	8	1	–	–
	$(\mu+\lambda)$-CGP (Subgraph)	2000	1	2.5	2.5	50	8	1	–	–
	Real-valued-CGP	2000	1	–	–	50	–	–	5	2
	Adaptive Real-valued-CGP	2000	[1,10]	–	–	[10,50]	–	–	5	2
Mult.-2Bit	$(1+4)$-CGP	1500	1	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	1500	1	–	–	–	–	1	–	–
	$(1+\lambda)$-CGP-ID	1500	1	5.0	5.0	–	–	1	–	–
	Canonical-CGP (Subgraph)	1500	1	–	–	20	–	–	5	2
	Canonical-CGP (Block)	1500	1	–	–	20	–	–	5	2
	$(\mu+\lambda)$-CGP (Subgraph)	1500	1	–	–	25	4	1	–	–
	$(\mu+\lambda)$-CGP (Block)	1500	1	–	–	25	4	1	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	1500	1	5.0	5.0	50	4	1	–	–
	Real-valued-CGP	1500	1	–	–	20	–	–	5	2
	Adaptive Real-valued-CGP	1500	[1,10]	–	–	[10,50]	–	–	5	2
Mult.-3Bit	$(1+4)$-CGP	2000	1	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	2000	1	–	–	–	–	1	–	–
	$(1+\lambda)$-CGP-ID	2000	1	0.5	0.5	–	–	1	–	–
	Canonical-CGP (Subgraph)	2000	1	–	–	20	–	–	5	2
	Canonical-CGP (Block)	2000	1	–	–	20	–	–	5	2
	$(\mu+\lambda)$-CGP (Subgraph)	2000	1	–	–	25	16	1	–	–
	$(\mu+\lambda)$-CGP (Block)	2000	1	–	–	25	16	1	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	2000	1	0.3	0.3	25	16	1	–	–
	Real-valued-CGP	2000	1	–	–	50	–	–	5	2
	Adaptive Real-valued-CGP	2000	[1,10]	–	–	[10,50]	–	–	5	2
Subtr.-2Bit	$(1+4)$-CGP	2000	1	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	2000	1	–	–	–	–	1	–	–
	$(1+\lambda)$-CGP-ID	2000	1	10.0	10.0	–	–	1	–	–
	Canonical-CGP (Subgraph)	2000	1	–	–	20	–	–	5	2
	Canonical-CGP (Subgraph)	2000	1	–	–	20	–	–	5	2
	$(\mu+\lambda)$-CGP (Subgraph)	2000	1	–	–	25	4	1	–	–
	$(\mu+\lambda)$-CGP (Block)	2000	1	–	–	25	4	1	–	–
	$(\mu+\lambda)$-CGP (Subgraph)	2000	1	5.0	5.0	25	4	1	–	–
	Real-valued-CGP	2000	1	–	–	20	–	–	5	2
	Adaptive Real-valued-CGP	2000	[1,10]	–	–	[10,50]	–	–	5	2

Table 9.10: Results of the meta optimization for the symbolic regression problems.

Problem	**Algorithm**	N	M_p	M_i	M_d	C	μ	λ	P	T
Koza-1	$(1+4)$-CGP	10	20	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	10	20	–	–	–	–	8	–	–
	$(1+\lambda)$-CGP-ID	10	20	2.5	2.5	–	–	8	–	–
	Canonical-CGP (Subgraph)	10	20	–	–	70	–	–	50	4
	Canonical-CGP (Block)	10	10	–	–	70	–	–	50	4
	$(\mu+\lambda)$-CGP (Subgraph)	10	20	–	–	70	4	16	–	–
	$(\mu+\lambda)$-CGP (Block)	10	20	–	–	70	4	16	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	10	20	5.0	5.0	70	4	16	–	–
	Real-valued-CGP	10	20	–	–	70	–	–	50	7
	Adaptive Real-valued-CGP	10	[1,20]	–	–	[50,90]	–	–	50	7
Koza-2	$(1+4)$-CGP	10	20	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	10	20	–	–	–	–	8	–	–
	$(1+\lambda)$-CGP-ID	10	20	2.5	2.5	–	–	8	–	–
	Canonical-CGP (Subgraph)	10	20	–	–	70	–	–	50	4
	Canonical-CGP (Block)	10	10	–	–	70	–	–	50	4
	$(\mu+\lambda)$-CGP (Subgraph)	10	20	–	–	90	4	16	–	–
	$(\mu+\lambda)$-CGP (Block)	10	20	–	–	70	4	16	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	10	20	5.0	5.0	70	4	16	–	–
	Real-valued-CGP	10	20	–	–	70	–	–	50	7
	Adaptive Real-valued-CGP	10	[1,20]	–	–	[50,90]	–	–	50	7
Koza-3	$(1+4)$-CGP	10	20	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	10	20	–	–	–	–	8	–	–
	$(1+\lambda)$-CGP-ID	10	20	2.5	2.5	–	–	8	–	–
	Canonical-CGP (Subgraph)	10	20	–	–	70			50	4
	Canonical-CGP (Block)	10	10	–	–	70	–	–	50	4
	$(\mu+\lambda)$-CGP (Subgraph)	10	20	–	–	70	1	8	–	–
	$(\mu+\lambda)$-CGP (Block)	10	20	–	–	70	4	16	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	10	20	5.0	5.0	70	4	16	–	–
	Real-valued-CGP	10	20	–	–	90	–	–	50	7
	Adaptive Real-valued-CGP	10	[1,20]	–	–	[50,90]	–	–	50	7
Nguyen-4	$(1+4)$-CGP	120	10	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	100	10	–	–	–	–	16	–	–
	$(1+\lambda)$-CGP-ID	100	10	10.0	10.0	–	–	16	–	–
	Canonical-CGP (Subgraph)	220	9	–	–	90	–	–	50	5
	Canonical-CGP (Block)	100	5	–	–	90	–	–	50	7
	$(\mu+\lambda)$-CGP (Subgraph)	200	1	–	–	70	10	200	–	–
	$(\mu+\lambda)$-CGP (Block)	100	10	–	–	90	10	20	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	100	10	10.0	10.0	90	10	20	–	–
	Real-valued-CGP	300	5	–	–	70	–	–	50	7
	Adaptive Real-valued-CGP	300	[1,10]	–	–	[20,90]	–	–	50	7
Nguyen-5	$(1+4)$-CGP	60	7	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	60	7	–	–	–	–	16	–	–
	$(1+\lambda)$-CGP-ID	60	7	10.0	10.0	–	–	16	–	–
	Canonical-CGP (Subgraph)	100	6	–	–	25	–	–	10	2
	Canonical-CGP (Block)	60	7	–	–	90	–	–	50	7
	$(\mu+\lambda)$-CGP (Subgraph)	300	5	–	–	70	10	250	–	–
	$(\mu+\lambda)$-CGP (Block)	60	5	–	–	75	10	20	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	60	7	5.0	5.0	90	4	16	–	–
	Real-valued-CGP	300	5	–	–	70	–	–	50	7
	Adaptive Real-valued-CGP	300	[1,10]	–	–	[10,90]	–	–	50	7
Nguyen-6	$(1+4)$-CGP	100	10	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	100	10	–	–	–	–	16	–	–
	$(1+\lambda)$-CGP-ID	100	7	10.0	10.0	–	–	16	–	–
	Canonical-CGP (Subgraph)	20	20	–	–	90	–	–	50	7
	Canonical-CGP (Block)	60	7	–	–	90	–	–	50	7
	$(\mu+\lambda)$-CGP (Subgraph)	300	1	–	–	70	10	250	–	–
	$(\mu+\lambda)$-CGP (Block)	60	7	–	–	90	4	16	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	60	7	5.0	5.0	90	4	16	–	–
	Real-valued-CGP	300	5	–	–	70	–	–	50	7
	Adaptive Real-valued-CGP	300	[1,10]	–	–	[10,90]	–	–	50	7
Nguyen-7	$(1+4)$-CGP	200	2	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	1000	2	–	–	–	–	16	–	–
	$(1+\lambda)$-CGP-ID	200	5	10.0	10.0	–	–	16	–	–
	Canonical-CGP (Subgraph)	500	3	–	–	70	–	–	250	7
	Canonical-CGP (Block)	200	5	–	–	90	–	–	50	7
	$(\mu+\lambda)$-CGP (Subgraph)	200	10	–	–	90	10	200	–	–
	$(\mu+\lambda)$-CGP (Block)	200	5	–	–	75	4	16	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	200	5	5.0	5.0	75	4	16	–	–
	Real-valued-CGP	200	5	–	–	70	–	–	50	7
	Adaptive Real-valued-CGP	300	[1,10]	–	–	[10,90]	–	–	50	7
Keijzer-6	$(1+4)$-CGP	2000	3	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	2000	5	–	–	–	–	16	–	–
	$(1+\lambda)$-CGP-ID	200	5	5.0	5.0	–	–	16	–	–
	Canonical-CGP (Subgraph)	100	5	–	–	70	–	–	250	10
	Canonical-CGP (Block)	200	5	–	–	70	–	–	50	7
	$(\mu+\lambda)$-CGP (Subgraph)	700	5	–	–	70	10	250	–	–
	$(\mu+\lambda)$-CGP (Block)	200	5	–	–	75	4	16	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	200	5	5.0	5.0	75	4	16	–	–
	Real-valued-CGP	2000	1	–	–	70	–	–	250	8
	Adaptive Real-valued-CGP	300	[1,10]	–	–	[10,90]	–	–	50	7
Pagie-1	$(1+4)$-CGP	1500	7	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	1500	7	–	–	–	–	16	–	–
	$(1+\lambda)$-CGP-ID	1500	1	20.0	20.0	–	–	16	–	–
	Canonical-CGP (Subgraph)	500	5	–	–	90	–	–	200	7
	Canonical-CGP (Block)	500	5	–	–	70	–	–	50	7
	$(\mu+\lambda)$-CGP (Subgraph)	500	8	–	–	75	25	125	–	–
	$(\mu+\lambda)$-CGP (Block)	500	5	–	–	75	4	16	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	500	5	20.0	20.0	75	4	16	–	–
	Real-valued-CGP	300	5	–	–	70	–	–	50	7
	Adaptive Real-valued-CGP	300	[1,10]	–	–	[20,70]	–	–	50	[4,10]

Table 9.11: Results of the meta evolution for the image operator design problems.

Problem	Algorithm	N	M_p	M_i	M_d	C	μ	λ	P	T
Salt & Pepper noise	$(1+4)$-CGP	3000	1	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	3000	1	–	–	–	–	1	–	–
	$(1+\lambda)$-CGP-ID	3000	1	5.0	5.0	–	–	–	–	–
	Canonical-CGP (Subgraph)	1000	1	–	–	50	–	–	10	2
	Canonical-CGP (Block)	3000	1	–	–	50	–	–	5	2
	$(\mu+\lambda)$-CGP (Subgraph)	3000	1	–	–	50	4	1	–	–
	$(\mu+\lambda)$-CGP (Block)	3000	1	–	–	50	4	1	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	3000	1	5.0	5.0	50	4	1	–	–
	Real-valued-CGP	3000	1	–	–	50	–	–	5	2
	Adaptive Real-valued-CGP	3000	[1,5]	–	–	[10,90]	–	–	5	2
Gaussian noise	$(1+4)$-CGP	3000	1	–	–	–	–	–	–	–
	$(1+\lambda)$-CGP	3000	1	–	–	–	–	1	–	–
	$(1+\lambda)$-CGP-ID	3000	1	5.0	5.0	–	–	–	–	–
	Canonical-CGP (Subgraph)	3000	1	–	–	50	–	–	5	2
	Canonical-CGP (Block)	3000	1	–	–	50	–	–	5	2
	$(\mu+\lambda)$-CGP (Subgraph)	3000	1	–	–	25	4	1	–	–
	$(\mu+\lambda)$-CGP (Block)	3000	1	–	–	50	4	1	–	–
	$(\mu+\lambda)$-CGP-ID (Subgraph)	3000	1	5.0	5.0	50	4	1	–	–
	Real-valued-CGP	3000	1	–	–	50	–	–	5	2
	Adaptive Real-valued-CGP	3000	[1,5]	–	–	[10,90]	–	–	5	2

number of fitness evaluations for all tested Boolean functions. This setting is also effective for the use of various setups of the $(\mu+\lambda)$-CGP. Overall, it can also be seen that the proposed advanced methods of crossover and mutation lead to a significant decrease of the number fitness evaluations to find the ideal solution.
Table 9.16 shows the results for the image operator design problems, and it can be seen that in this problem domain, the Canonical-CGP, the $(\mu+\lambda)$-CGP and the $(1+\lambda)$-CGP-ID perform best.

9.2.5 Conclusion

Our experiments show that the $(1+4)$-CGP and $(1+\lambda)$-CGP cannot be considered and generalized as the most effective ways to use CGP. However, in the Boolean domain, our results regarding the parametrization are predominantly coherent with the findings of Miller et al. [99]. Our experiments indicate that for the majority of our tested problems, the well-known CGP performance dogma of low population sizes and extremely high levels of redundancy is coherent with previous findings. However, our experiments on the 1-Bit and 2-Bit Adder also indicate that this dogma cannot be generalized in this problem domain. Since it has been found that genotypic crossover methods do not contribute to the performance of CGP [97] in this problem domain, our experiments show that the proposed phenotypic crossover and mutation operators can contribute significantly to the search performance by using a canonical and $(\mu+\lambda)$-EA. Consequently,the results indicate that the predominance of the $(1+4)$-CGP and $(1+\lambda)$-CGP algorithms cannot be generalized. Our experiments also confirm findings of the former work in the Boolean domain which has been presented in Chapter 5 and demonstrated that a setting of $\lambda = 1$ works more effective for the $(1+\lambda)$-CGP as a choice of $\lambda = 4$. Our results expand this finding to the recombination based $(\mu+\lambda)$-CGP for which a setting of $\lambda = 1$ led to good results in the Boolean domain.

In the symbolic regression domain, the experiments indicate that the use of the use of crossover can significantly contribute to the search performance. Especially, the

Table 9.12: Results of the algorithm comparison for the problems Koza 1,2,3 evaluated by the number of fitness evaluations (FE) to termination.

Problem	**Algorithm**	**Mean FE**	**SD**	**SEM**	**1Q**	**Median**	**3Q**	**Unfinished runs**
Koza-1	(1 + 4)-CGP	8675635	16681422	±1668142	441477	1814344	7045961	2
	(1 + λ)-CGP	7370880‡	17384354	±1738435	204400	1050936	4294170	3
	(1 + λ)-CGP-ID	7120871	14884170	±1488417	315530	1387696	4958900	0
	Canonical-CGP (Subgraph)	663822‡	838546	±83854	135162	337950	710275	0
	Canonical-CGP (Block)	**243837**‡	**474790**	**±47479**	**13046**	**82124**	**261084**	0
	(μ + λ)-CGP (Subgraph)	7780751‡	15830735	±1583073	197284	1830312	6318740	3
	(μ + λ)-CGP (Block)	50188221	101041195	±10104119	1489364	16759132	61052140	8
	(μ + λ)-CGP-ID (Subgraph)	9107437	22769750	±2276975	423124	1507108	7437168	0
	Real-valued-CGP	2650475‡	3665064	±366506	580825	1631850	3114612	0
	Adaptive Real-valued-CGP	1563294†	1569972	±156997	468329	1008885	2219834	0
Koza-2	(1 + 4)-CGP	8264426	19894512	±1989451	150140	888884	4378756	6
	(1 + λ)-CGP	8191549	20275790	±2027579	94290	559028	4710848	1
	(1 + λ)-CGP-ID	1094562†	1855864	±185586	27842	372728	1282919	0
	Canonical-CGP (Subgraph)	444118‡	95000	±286700	627550	29650	78800	0
	Canonical-CGP (Block)	**114138**‡	**178463**	**±17846**	**12862**	**40523**	**141022**	0
	(μ + λ)-CGP (Subgraph)	5729778	11021660	±1102166.	238156	1320880	5878696	1
	(μ + λ)-CGP (Block)	22653191	54817816	±5481781	738020	3531700	19806332	3
	(μ + λ)-CGP-ID (Subgraph)	5727207	13465035	±1346503	148248	949988	5286208	3
	Real-valued-CGP	788450†	885517	±88551	138987	499000	1094325	0
	Adaptive Real-valued-CGP	698299‡	809171	±80917	152990	455357	920734	0
Koza-3	(1 + 4)-CGP	600153	1214527	±121452	39076	177418	443038	0
	(1 + λ)-CGP	753551	2535215	±253521	29528	120368	431318	0
	(1 + λ)-CGP-ID	422651†	949911	±94991	11520	77080	304820	0
	Canonical-CGP (Subgraph)	**32870**‡	**57156**	**±10435**	**2488**	**6700**	**32713**	0
	Canonical-CGP (Block)	70795‡	216043	±21604	6161	15190	43414	0
	(μ + λ)-CGP (Subgraph)	926857	3473467	±347347	28548	121040	362180	0
	(μ + λ)-CGP (Block)	1167707	2287317	±228731	58832	220588	1150756	0
	(μ + λ)-CGP-ID (Subgraph)	738653	3622910	±362291	33460	109700	386732	0
	Real-valued-CGP	124457‡	183825	±18382	23737	60950	144262	0
	Adaptive Real-valued-CGP	119034‡	213922	±21392	15863	43144	110580	0

Table 9.13: Results of the algorithm comparison algorithm for the symbolic regression problems evaluated with the *best-fitness-of-run* method.

Problem	Algorithm	Mean Best Fitness	SD	SEM	1Q	Median	3Q
Nguyen-4	$(1+4)$-CGP	0.68	0.55	±0.05	0.34	0.58	0.77
	$(1+\lambda)$-CGP	0.61	0.46	±0.04	0.35	0.54	0.74
	$(1+\lambda)$-CGP-ID	**0.51**‡	**0.39**	**±0.03**	**0.22**	**0.42**	**0.67**
	Canonical-CGP (Subgraph)	**0.50**†	**0.28**	**±0.04**	**0.31**	**0.47**	**0.60**
	Canonical-CGP (Block)	0.62	0.55	±0.05	0.30	0.54	0.63
	$(\mu+\lambda)$-CGP (Subgraph)	0.60†	0.40	±0.04	0.36	0.54	0.76
	$(\mu+\lambda)$-CGP (Block)	0.88	0.66	±0.07	0.53	0.67	1.01
	$(\mu+\lambda)$-CGP-ID (Subgraph)	0.55	0.34	±0.03	0.31	0.54	0.74
	Real-valued-CGP	0.71	0.75	±0.07	0.29	0.55	0.87
	Adaptive Real-valued-CGP	0.53†	0.37	±0.04	0.27	0.47	0.65
Nguyen-5	$(1+4)$-CGP	0.45	0.42	±0.04	0.06	0.32	0.81
	$(1+\lambda)$-CGP	0.39	0.33	±0.03	0.08	0.27	0.63
	$(1+\lambda)$-CGP-ID	**0.21**‡	**0.20**	**±0.02**	**0.04**	**0.12**	**0.36**
	Canonical-CGP (Subgraph)	0.29‡	0.27	±0.03	0.05	0.20	0.40
	Canonical-CGP (Block)	0.34	0.33	±0.03	0.05	0.26	0.59
	$(\mu+\lambda)$-CGP (Subgraph)	0.28‡	0.25	±0.02	0.06	0.19	0.45
	$(\mu+\lambda)$-CGP (Block)	0.48	0.40	±0.04	0.15	0.41	0.82
	$(\mu+\lambda)$-CGP-ID (Subgraph)	0.29†	0.25	±0.02	0.09	0.24	0.40
	Real-valued-CGP	0.30‡	0.24	±0.02	0.10	0.25	0.47
	Adaptive Real-valued-CGP	0.31†	0.26	±0.02	0.12	0.24	0.43
Nguyen-6	$(1+4)$-CGP	0.54	0.66	±0.06	0.16	0.29	0.61
	$(1+\lambda)$-CGP	0.50	0.67	±0.06	0.15	0.22	0.50
	$(1+\lambda)$-CGP-ID	0.35	0.33	±0.03	0.15	0.26	0.44
	Canonical-CGP (Subgraph)	**0.31**‡	**0.31**	**±0.03**	**0.15**	**0.24**	**0.40**
	Canonical-CGP (Block)	0.44	0.63	±0.06	0.14	0.24	0.47
	$(\mu+\lambda)$-CGP (Subgraph)	0.61	0.67	±0.06	0.16	0.35	0.67
	$(\mu+\lambda)$-CGP (Block)	0.83	0.85	±0.08	0.23	0.38	1.28
	$(\mu+\lambda)$-CGP-ID (Subgraph)	0.34†	0.37	±0.03	0.13	0.22	0.42
	Real-valued-CGP	0.76	0.82	±0.08	0.17	0.37	0.98
	Adaptive Real-valued-CGP	0.58	0.72	±0.07	0.17	0.31	0.56
Nguyen-7	$(1+4)$-CGP	0.79	0.48	±0.05	0.45	0.67	1.06
	$(1+\lambda)$-CGP	0.71	0.45	±0.04	0.44	0.67	0.76
	$(1+\lambda)$-CGP-ID	0.63†	0.36	±0.03	0.39	0.59	0.68
	Canonical-CGP (Subgraph)	**0.60**‡	**0.35**	**±0.03**	**0.36**	**0.60**	**0.68**
	Canonical-CGP (Block)	0.72	0.52	±0.05	0.47	0.65	0.70
	$(\mu+\lambda)$-CGP (Subgraph)	0.62‡	0.40	±0.04	0.42	0.63	0.68
	$(\mu+\lambda)$-CGP (Block)	1.39	1.13	±0.11	0.67	0.97	1.73
	$(\mu+\lambda)$-CGP-ID (Subgraph)	0.64‡	0.48	±0.04	0.36	0.60	0.70
	Real-valued-CGP	0.82	0.61	±0.06	0.50	0.67	0.88
	Adaptive Real-valued-CGP	0.71	0.51	±0.05	0.43	0.67	0.69
Keijzer-6	$(1+4)$-CGP	3.78	2.61	±0.26	2.16	3.24	4.59
	$(1+\lambda)$-CGP	3.38	2.52	±0.25	2.41	3.03	3.158
	$(1+\lambda)$-CGP-ID	3.48‡	2.61	±0.26	1.91	2.96	3.96
	Canonical-CGP (Subgraph)	**2.81**†	**1.13**	**±0.11**	**1.78**	**2.90**	**3.75**
	Canonical-CGP (Block)	3.71	2.28	±0.22	2.37	3.15	4.01
	$(\mu+\lambda)$-CGP (Subgraph)	2.88†	1.09	±0.10	2.25	3.14	3.15
	$(\mu+\lambda)$-CGP (Block)	5.07	3.69	±0.36	3.16	3.94	5.46
	$(\mu+\lambda)$-CGP-ID (Subgraph)	3.70	2.98	±0.29	1.87	2.85	4.03
	Real-valued-CGP	3.97	2.56	±0.25	3.08	3.22	4.15
	Adaptive Real-valued-CGP	6.58	6.73	±0.67	2.82	4.40	6.55
Pagie-1	$(1+4)$-CGP	128.18	48.19	±4.81	87.81	119.09	161.08
	$(1+\lambda)$-CGP	120.75	44.95	±4.49	86.14	120.91	155.06
	$(1+\lambda)$-CGP-ID	109.23‡	35.15	±3.51	83.35	104.35	136.12
	Canonical-CGP (Subgraph)	**98.52**‡	**50.57**	**±5.08**	**59.04**	**85.31**	**130.04**
	Canonical-CGP (Block)	119.97	45.44	±4.54	82.71	115.41	151.11
	$(\mu+\lambda)$-CGP (Subgraph)	99.74‡	41.25	±4.12	65.32	95.79	131.76
	$(\mu+\lambda)$-CGP (Block)	160.15	45.71	±4.57	133.84	157.33	187.67
	$(\mu+\lambda)$-CGP-ID (Subgraph)	119.96	44.64	±4.46	91.21	112.94	146.48
	Real-valued-CGP	112.75†	47.15	±4.71	72.54	108.48	145.40
	Adaptive Real-valued-CGP	109.24	47.85	±4.78	72.48	95.87	143.32

Table 9.14: Results of the algorithm comparison for the single output Booleans function problem evaluated by the number of fitness evaluations (FE) to termination.

Problem	Algorithm	Mean FE	SD	SEM	1Q	Median	3Q
Parity-3	$(1+4)$-CGP	3194	3428	±342	1324	2144	3756
	$(1+\lambda)$-CGP	2175‡	1836	±183	857	1649	2868
	$(1+\lambda)$-CGP-ID	**1560**‡	**1181**	**±118**	**740**	**1093**	**1906**
	Canonical-CGP (Subgraph)	3567	3356	±335	1539	2378	4455
	Canonical-CGP (Block)	2977	2239	±223	1183	2640	3873
	$(\mu+\lambda)$-CGP (Subgraph)	1808‡	1670	±167	649	1301	2119
	$(\mu+\lambda)$-CGP (Block)	1781‡	1432	±143	739	1265	2509
	$(\mu+\lambda)$-CGP-ID (Block)	**1340**‡	**965**	**±96**	**708**	**1125**	**1874**
	Real-valued-CGP	2089‡	1619	±161	886	1600	2760
	Adaptive Real-valued-CGP	2162‡	1954	±1954	867	1561	2741
Parity-4	$(1+4)$-CGP	15420	14152	±1422	6292	10358	17726
	$(1+\lambda)$-CGP	9615‡	6967	±696	4798	7703	13391
	$(1+\lambda)$-CGP-ID	**6206**‡	**3601**	**±360**	**3803**	**5493**	**7298**
	Canonical-CGP (Subgraph)	26456	37024	±3702	10397	16526	29247
	Canonical-CGP (Block)	24463	23404	±2340	11158	17742	26914
	$(\mu+\lambda)$-CGP (Subgraph)	9211‡	7349	±735	4798	7392	11099
	$(\mu+\lambda)$-CGP (Block)	12202	8611	±861	6030	9593	15992
	$(\mu+\lambda)$-CGP-ID (Subgraph)	**6218**‡	**3962**	**±396**	**3505**	**5284**	**7768**
	Real-valued-CGP	11462	10727	±1072	6572	12212	197825
	Adaptive Real-valued-CGP	15491	12818	±1282	6438	11271	21084
Parity-5	$(1+4)$-CGP	45542	33947	±3411	21524	36834	61222
	$(1+\lambda)$-CGP	32248‡	24228	±2422	17129	26249	39767
	$(1+\lambda)$-CGP-ID	25920‡	15363	±1536	14954	23128	31118
	Canonical-CGP (Subgraph)	92641‡	61294	±6129	55893	77522	103291
	Canonical-CGP (Block)	102803	111238	±11123	47691	76170	122404
	$(\mu+\lambda)$-CGP (Subgraph)	30781‡	23159	±2315	16315	24064	39778
	$(\mu+\lambda)$-CGP (Block)	32298‡	18885	±1888	19514	28927	43804
	$(\mu+\lambda)$-CGP-ID (Subgraph)	**25142**‡	**14411**	**±1441**	**15141**	**22082**	**30505**
	Real-valued-CGP	66994	37566	±6858	38518	57028	91524
	Adaptive Real-valued-CGP	99885	82628	±8262	48619	75104	118023
Parity-6	$(1+4)$-CGP	199989	142915	±14291	107418	163234	242573
	$(1+\lambda)$-CGP	92506‡	62113	±6211	45346	77795	117251
	$(1+\lambda)$-CGP-ID	**73722**‡	**41065**	**±4106**	**43564**	**63833**	**87546**
	Canonical-CGP (Subgraph)	242986	161762	±16257	134518	200196	309346
	Canonical-CGP (Block)	327097	218152	±21815	167538	286582	421521
	$(\mu+\lambda)$-CGP (Subgraph)	82428‡	50537	±5053	48087	68923	97539
	$(\mu+\lambda)$-CGP (Block)	80330‡	42173	±4217	50084	71790	96113
	$(\mu+\lambda)$-CGP-ID (Subgraph)	**74848**‡	**53501**	**±5350**	**38799**	**58689**	**93726**
	Real-valued-CGP	239409	161504	±16150	140885	202612	275401
	Adaptive Real-valued-CGP	237918	165152	±16515	139388	197367	296735
Parity-7	$(1+4)$-CGP	480055	301612	±30161	268210	393362	605382
	$(1+\lambda)$-CGP	246242‡	152576	±15257	124592	186627	280166
	$(1+\lambda)$-CGP-ID	225357‡	152961	±15296	123022	185180	280166
	Canonical-CGP (Subgraph)	631568	548180	±54818	293613	453204	750792
	Canonical-CGP (Block)	850030	601124	±60112	452679	696466	1075600
	$(\mu+\lambda)$-CGP (Subgraph)	**201532**‡	**131936**	**±13193**	**102515**	**172038**	**247155**
	$(\mu+\lambda)$-CGP (Block)	263613‡	179168	±17916	128220	215078	339860s
	$(\mu+\lambda)$-CGP-ID (Subgraph)	**208606**‡	**141268**	**±14126**	**111079**	**170310**	**254892**
	Real-valued-CGP	717328	526545	±52654	435585	590590	904177
	Adaptive Real-valued-CGP	591460	372779	±37277	282502	531208	800381
Parity-8	$(1+4)$-CGP	1450643	944550	±172450	933673	1224538	1883852
	$(1+\lambda)$-CGP	522204‡	346562	±63273	301194	410303	604323
	$(1+\lambda)$-CGP-ID	621120‡	767796	±140179	224878	420112	696398
	Canonical-CGP (Subgraph)	1594115	1246555	±227588	935212	1152674	1675549
	Canonical-CGP (Block)	2601042	1645616	±300447	1486101	2024344	3526173
	$(\mu+\lambda)$-CGP (Subgraph)	**465055**‡	**280553**	**±51221**	**268547**	**378473**	**565339**
	$(\mu+\lambda)$-CGP (Block)	614304‡	276234	±50433	462495	565943	669591
	$(\mu+\lambda)$-CGP-ID (Subgraph)	560684‡	460684	±84109	315760	396262	628762
	Real-valued-CGP	1624972	797217	±145551	1014858	1542978	1929133
	Adaptive Real-valued-CGP	1577949	809408	±147777	1082068	1445260	1973013

Table 9.15: Results of the algorithm comparison for the multiple output Boolean function problems evaluated by the number of fitness evaluations (FE) to termination.

Problem	**Algorithm**	**Mean FE**	**SD**	**SEM**	**1Q**	**Median**	**3Q**
Adder-1Bit	(1 + 4)-CGP	8373	9847	±984	2892	4826	10270
	(1 + λ)-CGP	4697‡	4528	±452	1996	3024	5440
	(1 + λ)-CGP-ID	4048‡	3645	±364	1627	3121	5032
	Canonical-CGP (Subgraph)	12296	129264	±1292	4194	7980	16291
	Canonical-CGP (Block)	12912	15265	±1526	3838	7734	14993
	(μ + λ)-CGP (Subgraph)	5193‡	4877	±487	2148	3699	6143
	(μ + λ)-CGP (Block)	5564‡	4867	±486	1962	3900	7177
	(μ + λ)**-CGP-ID (Subgraph)**	**3896**‡	**3112**	**±312**	**1947**	**2702**	**5508**
	Real-valued-CGP	12833	13148	±1314	3594	8960	16779
	Adaptive Real-valued-CGP	12164	11740	±1174	5867	8036	14862
Adder-2Bit	(1 + 4)-CGP	146540	138275	±13827	68661	105746	176126
	(1 + λ)-CGP	86168‡	84130	±8413	35658	58135	104502
	(1 + λ)-CGP-ID	83629‡	81815	±8181	33988	64709	92365
	Canonical-CGP (Subgraph)	470193	380088	±38008	193837	329472	649081
	Canonical-CGP (Block)	232328	255735	±25573	90993	191414	286934
	(μ + λ)**-CGP (Subgraph)**	**73312**‡	**62167**	**±6216**	**32654**	**50795**	**105657**
	(μ + λ)-CGP (Block)	96382‡	94841	±9484	38355	63795	116310
	(μ + λ)-CGP-ID (Subgraph)	84546‡	73079	±7307	29579	59428	125185
	Real-valued-CGP	224422	202668	±20266	92722	146868	312855
	Adaptive Real-valued-CGP	210683	179114	±17911	84732	170680	249162
Adder-3Bit	(1 + 4)-CGP	628775	508883	±50888	315038	471514	832283
	(1 + λ)-CGP	293863‡	231491	±23149	143663	223511	415815
	(1 + λ)**-CGP-ID**	**271797**‡	**209510**	**±20951**	**149857**	**218817**	**327394**
	Canonical-CGP	1023690	821377	±82137	494814	771332	1289180
	Canonical-CGP (Block)	823207	568404	±56840	447645	687500	1031797
	(μ + λ)**-CGP (Subgraph)**	**269503**‡	**156087**	**±15608**	**141141**	**252469**	**352199**
	(μ + λ)-CGP (Block)	358237‡	275076	±27507	204995	281461	401565
	(μ + λ)-CGP-ID (Subgraph)	291382‡	191363	±19136	172473	228554	361305
	Real-valued-CGP	1026429	668030	±66803	561160	893494	1228132
	Adaptive Real-valued-CGP	2177706	1819147	±183495	1079543	1666680	2611373
Multiplier-2Bit	(1 + 4)-CGP	13535	21488	±2148	4382	7590	13440
	(1 + λ)-CGP	7381‡	7135	±713	3210	4988	8737
	(1 + λ)**-CGP-ID**	**6207**‡	**4548**	**±454**	**3137**	**4998**	**7515**
	Canonical-CGP (Subgraph)	14624	12711	±1271	7457	10100	18421
	Canonical-CGP (Block)	19215	32654	±3265	7055	12276	18529
	(μ + λ)-CGP (Subgraph)	7118‡	9592	±959	3032	4833	8524
	(μ + λ)**-CGP (Block)**	**6246**‡	**4941**	**±494**	**3293**	**4812**	**7674**
	(μ + λ)-CGP-ID (Subgraph)	7091‡	8725	±872	3208	5147	7045
	Real-valued-CGP	20422	40342	±4034	10363	14342	19127
	Adaptive Real-valued-CGP	14604	10337	±1033	8041	11356	17278
Multiplier-3Bit	(1 + 4)-CGP	2115131	1728492	±172849	1179835	1660106	2557923
	(1 + λ)-CGP	1012076‡	563186	±56318	623751	893539	1254464
	(1 + λ)-CGP-ID	1050349‡	926718	±92671	603639	802820	1274927
	Canonical-CGP	3070470	1821561	±182156	1794079	2577410	3954159
	Canonical-CGP (Block)	3123525	1957714	±195771	1905498	2631162	3988024
	(μ + λ)**-CGP (Subgraph)**	**949663**‡	6**99647**	**±69964**	5**20692**	**721905**	**1084116**
	(μ + λ)-CGP (Block)	1136368‡	730285	±73028	636331	978148	1464708
	(μ + λ)-CGP-ID (Subgraph)	1158713‡	989079	±98907	590971	844934	1291924
	Real-valued-CGP	3937804	2469714	±246971	2158471	3419286	5116702
	Adaptive Real-valued-CGP	3493443	2192502	±219250	2030986	3004752	4442449
Subtractor-2Bit	(1 + 4)-CGP	22743	25293	±2529	8047	14404	27818
	(1 + λ)-CGP	15942‡	17449	±1744	4609	10235	17203
	(1 + λ)-CGP-ID	12906‡	11372	±1137	4950	9914	17154
	Canonical-CGP (Subgraph)	36895	42253	±4225	1241	19052	38932
	Canonical-CGP (Block)	35578	33030	±3303	16498	26910	40101
	(μ + λ)**-CGP (Subgraph)**	**13525**‡	**15077**	**±1507**	**5589**	**8474**	**15262**
	(μ + λ)-CGP (Block)	13198‡	7331	±733	7376	11916	18442
	(μ + λ)-CGP-ID (Subgraph)	15714‡	19471	±1947	5770	9347	15875
	Real-valued-CGP	27563	24590	±2459	12421	20618	31347
	Adaptive Real-valued-CGP	32408	37964	±3796	14500	21470	36388

Table 9.16: Results of the algorithm comparison for the image operator design problems evaluated by the number of fitness evaluations (FE) to termination.

Problem	**Algorithm**	**Mean**	**SD**	**SEM**	**1Q**	**Median**	**3Q**
		Fitness Evaluations					
Salt & Pepper noise	(1 + 4)-CGP	48983	140563	±14056	3274	7818	25259
	(1 + λ)-CGP	18985	39459	±3945	3280	6220	16979
	(1 + λ)-CGP-ID	**10406**‡	**25018**	**±2501**	**2554**	**4128**	**7269**
	Canonical-CGP (Subgraph)	57227	104822	±10482	7788	14124	44690
	Canonical-CGP (Block)	**10469**‡	**18697**	**±1869**	**2543**	**4672**	**9931**
	(μ + λ)-CGP (Subgraph)	52879	132183	±13218	4586	11294	40977
	(μ + λ)**-CGP (Block)**	**13169**‡	**36769**	**±3676**	**1956**	**3479**	**9145**
	(μ + λ)-CGP-ID (Subgraph)	40326	214992	±21499	4124	6340	10605
	Real-valued-CGP	123112	506007	±50600	3919	10390	28092
	Adaptive Real-valued-CGP	31069	83899	±8389	2987	7148	19862
Gaussian noise	(1 + 4)-CGP	58738	313996	±31399	4343	9202	22169
	(1 + λ)-CGP	24023	52737	±5273	3098	7776	17612
	(1 + λ)-CGP-ID	**5566**‡	**10289**	**±1028**	**1905**	**3384**	**5934**
	Canonical-CGP (Subgraph)	76775	117230	±11723	12332	30520	80268
	Canonical-CGP (Block)	33568	111036	±11103	3015	6828	16327
	(μ + λ)-CGP (Subgraph)	35768	80891	±8093	4765	10007	30636
	(μ + λ)-CGP (Block)	15680‡	43775	±4377	1619	4118	12664
	(μ + λ)-CGP-ID (Subgraph)	6155‡	5251	±525	2742	4329	8117
	Real-valued-CGP	24606‡	21007	±2100	10954	17300	32453
	Adaptive Real-valued-CGP	38455	107083	±10708	3095	6710	20697

performance of the Canonical-CGP equipped with subgraph crossover was superior to the $(1 + 4)$-CGP on all tested problems in this problem domain. Furthermore, our comparison of the whole evolutionary process for the more simple benchmark problems Koza 1,2 & 3 shows a big gap of the search performance between the (1+4)-CGP and the Canonical-CGP on these problems. Furthermore, the Canonical-CGP finished all runs successfully within the budget of 10^6 fitness evaluations. Our experiments in the symbolic regression domain revealed a contrary situation in which comparatively smaller genotypes and bigger populations perform more effective. This finding will be analyzed in more detail in the following sections.

Regarding the use of the *insertion* and *deletion* mutation, we observed an improved search performance in all three problem domains on all tested problems. The use of both techniques in combination with the subgraph crossover with a $(\mu + \lambda)$-EA was beneficial in several cases but turned out not to be the best choice in some cases. Furthermore, in some cases the use of the $(\mu + \lambda)$-EA with crossover and both mutations reduced the improvement of the search performance when compared to the results of the $(\mu + \lambda)$-EA without *insertion* and *deletion* mutation.

The following concluding remarks can be drawn from the results of the experiments.

- Advanced crossover and mutation operators can improve the search performance of CGP
- Bigger populations can be effectively used with CGP
- Well known parametrization patterns cannot be generalized

9.3 Analytic Algorithm Comparison

One question, which arises from the results of former chapters and former sections of this chapter is *"What are the reasons for the contrary situation in the Boolean function and the symbolic regression domain?"*. On one hand, it is well known that small population sizes perform effectively in the Boolean function domain. On the other hand, it is little known that medium and big population sizes perform effectively in the symbolic regression domain when CGP is used. In this section, we will present results that shed more light on this antithetical situation in CGP. Furthermore, there is also only a little knowledge about small population sizes perform more effective than middle or big population sizes in the Boolean domain. Consequently, we will start our analysis in the following subsection with an investigation of the population size in the Boolean domain. In this section, we will analyze the following formulated research questions:

Research Question. (Wasted fitness evaluations) *Does the use of middle-size and big populations in CGP lead to wasted fitness evaluations?*

Research Question. (Random Initialization vs. Point Mutation) *Cause random initialization and point mutation similar effects?*

Since bigger populations seem to be not very useful in the Boolean domain, we assume that the increase of the population-size also increases the number of wasted fitness evaluations. Furthermore, since CGP is usually used with comparatively high rates of point mutation (e.g. 4% mutation rate for 100 function nodes or 20% for 10 function nodes have frequently been used in former work), we also compare the effects of random initialization and point mutation and assess the significance for the analysis of the first question.

9.3.1 (1+4)-CGP vs. (1+λ)-CGP vs. Pop50

Population size comparison in the Boolean domain

The findings of the effectiveness of small population sizes in CGP are based on experiments with the popular parity even problems, and a set of Boolean multiple output problems. Therefore, we compared the (1+λ)-CGP algorithm to the (1+4)-CGP on a set of simple and well-known Boolean function problems such as the parity-even problem with 2, 3 and 4 bits as input vectors. To make comparisons with multiple output problems, we chose the 1-Bit digital adder and the 2-Bit digital multiplier & subtractor problems. We compared different settings of the (1+λ)-CGP algorithm. We parameterized the λ parameter with settings of 1, 2, 4, 8, 16, 32, 64, and 128 offspring. The purpose of our comparison was to analyze the trending graph of the search performance when the λ parameter is exponentially decreased. For the very simple problems of our tested problems such as the Parity-2 and Parity-3 problems, we chose a genome size of 10 function nodes and a point mutation rate of 20%. For the remaining more complex problems of our tested problems, we chose a genome size of 100 nodes and a point mutation rate of 4%. The complete configuration for our experiments in the Boolean domain is shown in Table 9.17. For the experiments in the symbolic regression domain, we used the problems Koza 1,2 & 3. For all three problems, we used 10 function nodes and a point mutation rate of 20%. To offer the statistical validity of our results, we performed 100 runs for each setting of the λ parameter. We also evaluated a canonical CGP algorithm with a population size of 50 individuals. We used the subgraph crossover and the standard point mutation for the so-called Pop-50 CGP algorithm. We chose a tournament selection size of 4 and an elitism size of 2. The canonical CGP algorithm is denoted as Pop-50. To evaluate the search performance of the CGP algorithms, we measured the number of fitness evaluations until the CGP algorithm successfully terminated. In addition to the mean values of the measurements, we calculated the standard deviation (SD) and the standard error of the mean (SEM). We also calculated the median and the first and third quartiles.

Table 9.17: Configuration of the CGP algorithm for the respective test problems.

Problem	**Number of function nodes**	**Point mutation rate [%]**	**Number of input nodes**	**Number of output nodes**
Parity-2	10	20	2	1
Parity-3	10	20	3	1
Parity-4	100	4	4	1
Adder-1Bit	100	4	3	2
Multiplier-2Bit	100	4	4	4
Subtractor-2Bit	100	4	4	3

Table 9.18: Results of the algorithm comparison for the Boolean function problems between the $(1+1)$-CGP and the Pop-50 CGP.

Problem	**Algorithm**	**Mean FE**	**SD**	**SEM**	**1Q**	**Median**	**3Q**
Parity-2	$(1+1)$	483	487	±48	128	334	653
	Pop-50	1103	1668	±166	144	384	1248
Parity-3	$(1+1)$	59401	63699	±6340	17179	39464	76624
	Pop-50	426091	475867	±47587	116220	282000	471096
Parity-4	$(1+1)$	20650	19581	±1958	8290	14564	27183
	Pop-50	1054456	1613799	±161380	174252	511080	1125588
Adder-1Bit	$(1+1)$	5155	5470	±547	1880	3632	6209
	Pop-50	389827	391812	±39181	140352	264336	445536
Multiplier-2Bit	$(1+1)$	15678	16828	±1683	6164	10475	16493
	Pop-50	3045833	3146575	±314657	1155696	2103312	3278208
Subtractor-2Bit	$(1+1)$	15678	16828	±1683	6164	10475	16493
	Pop-50	50025	43723	±4372	22079	36960	61226

Table 9.19: Results for various settings of the $(1+\lambda)$-CGP and the Pop-50 CGP on the symbolic regression problems.

Problem	**Algorithm**	**Mean Fitness Evaluation**	**SD**	**SEM**	**1Q**	**Median**	**3Q**
Koza-1	$(1+1)$	155026	492304	±49230	1397	5197	22633
	$(1+2)$	146919	446898	±44689	1588	7717	28498
	$(1+4)$	153653	497639	±49763	1239	4844	20268
	$(1+8)$	195063	540627	±54062	1752	7396	46046
	$(1+16)$	105361	405482	±40548	1396	3800	11936
	$(\mathbf{1+32})$	**48099**	**210511**	**±21051**	**1304**	**3472**	**13368**
	$(1+64)$	87439	297167	±29716	2640	6944	41536
	$(1+128)$	82219	329522	±32952	2048	6016	22464
	Pop-50	**9299**	**28479**	**±2847**	**900**	**1776**	**4884**
Koza-2	$(1+1)$	520804	770988	±77098	11769	86453	571046
	$(1+2)$	484069	721568	±72156	11207	80856	587281
	$(1+4)$	509258	764939	±76493	14739	63790	844896
	$(1+8)$	460586	719263	±71926	10966	81200	516898
	$(1+16)$	424124	700427	±70043	9652	55488	373488
	$(\mathbf{1+32})$	**426754**	**693033**	**±69303**	**8680**	**53664**	**354224**
	$(1+64)$	474576	694991	±69499	13744	80896	644704
	$(1+128)$	454301	703801	±70380	15904	69120	494784
	Pop-50	**171903**	**376590**	**±37659**	**4272**	**25440**	**177108**
Koza-3	$(1+1)$	468028	695299	±69530	8421	120182	593541
	$(1+2)$	488043	751159	±75115	4471	69472	580894
	$(1+4)$	429402	709740	±70974	7432	51708	377573
	$(1+8)$	435227	699703	±69970	6354	52608	515092
	$(\mathbf{1+16})$	**343591**	**601858**	**±60185**	**5676**	**34672**	**370888**
	$(1+32)$	413876	672136	±67214	5664	56032	379960
	$(1+64)$	551257	739185	±73919	26384	153536	807376
	$(1+128)$	451704	710599	±71060	11776	85440	486976
	Pop-50	**146122**	**318450**	**±31844**	**3396**	**13584**	**134916**

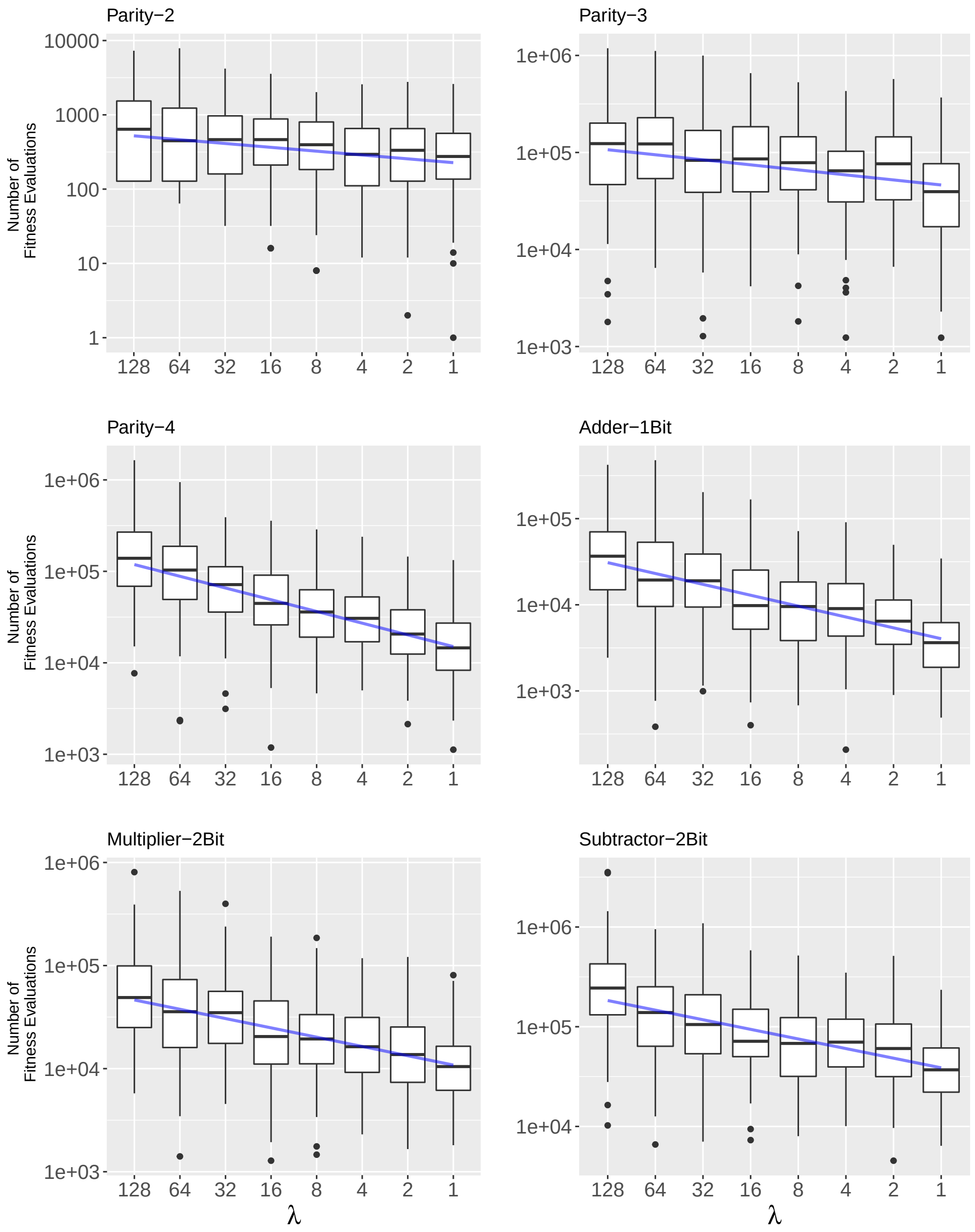


Figure 9.4: Results of the search performance comparison of the $(1 + \lambda)$-algorithm for various Boolean fucntion problems.

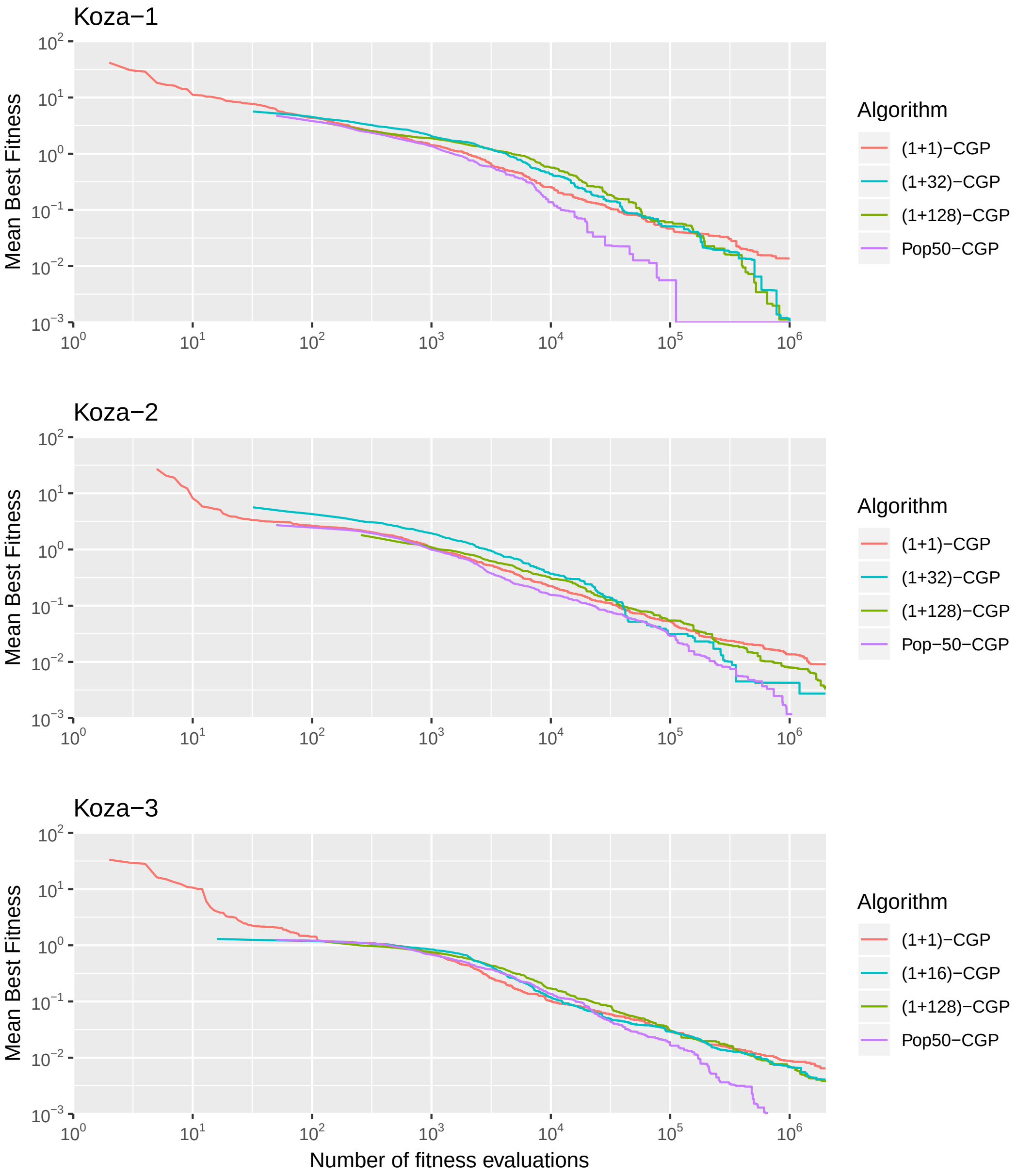


Figure 9.5: Convergence curves of the best individual of different settings of the (1+λ)-CGP and the Pop-50 CGP, covering the three symbolic regression problems Koza 1,2 and 3.

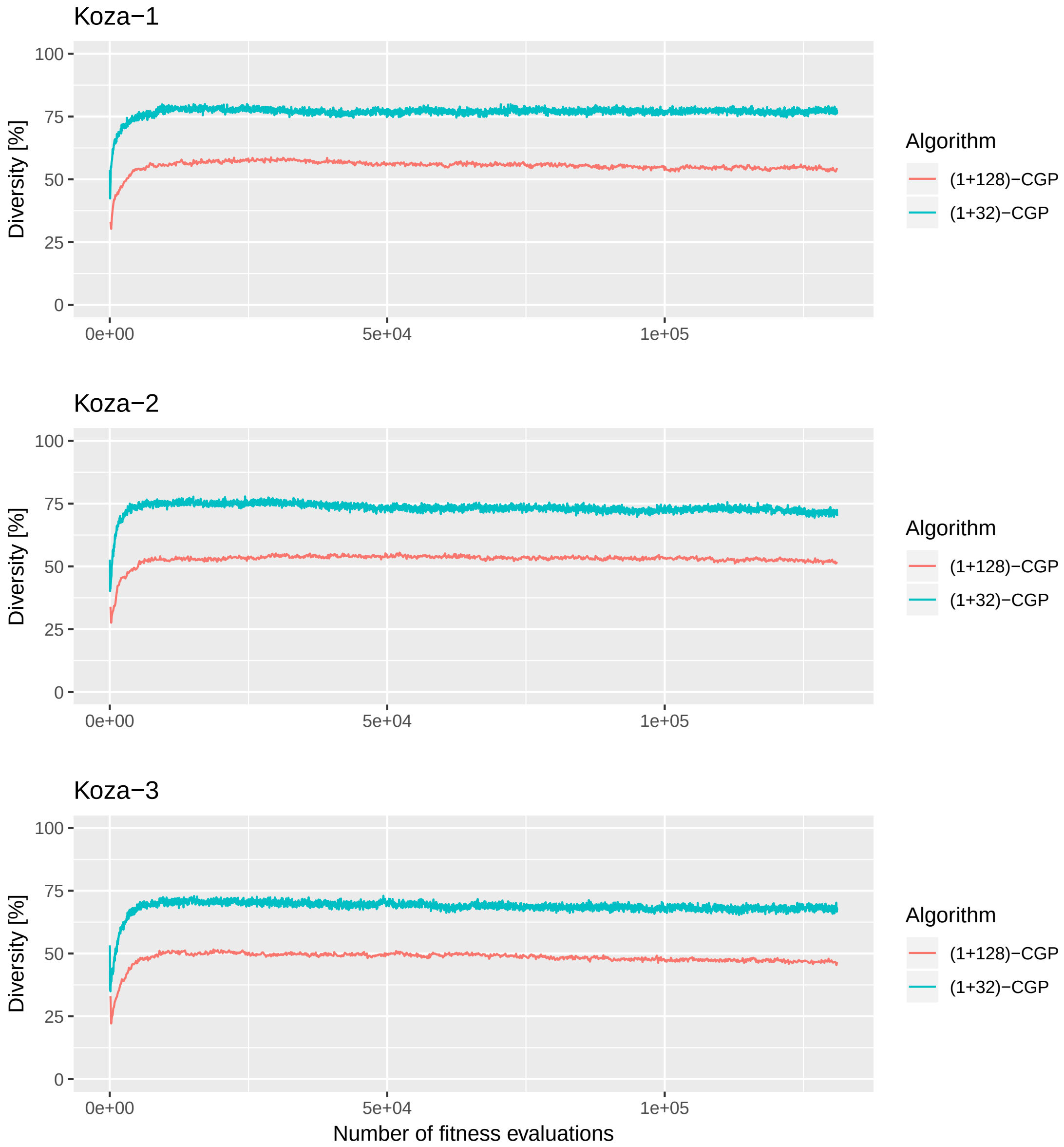


Figure 9.6: Diversity measurement over more than 10000 fitness evaluations of the (1+32)-CGP algoritm and the (1+128)-CGP for the three symbolic regression problems Koza 1,2 and 3.

Table 9.18 shows the results of the algorithm comparison in the Boolean function domain. The (1+λ)-CGP clearly outperforms the the Pop-50 CGP. Figure 9.4 shows the results of the comparison of the search performance with different settings of the λ parameter. The decrease of the λ parameter from 128 to 1 improves the search performance on every tested problem in this domain. Table 9.19 shows the results of the algorithm comparison in the symbolic regression domain. It is visible that the Pop-50 algorithm is superior to all tested settings of the $(1 + \lambda)$ algorithm on our tested problems. It can also be seen that the $(1 + \lambda)$-CGP performs effectively with a middle-size population. Figure 9.5 shows the convergence behavior of various settings of the λ parameter and the Pop-50 CGP algorithm. It can be seen that the use of the Pop-50-CGP leads to a much steeper convergence curve when compared to the curves of the (1+λ) algorithm. It can also be seen that the curves of the (1+16), (1+32), and (1+128) are steeper for the latter generations than the curves of the $(1 + 1)$-CGP. We will analyze and discuss the results of these experiments in the respective subsection for discussion.

Figure 9.6 shows the results of the diversity measurement on the three regression problems Koza 1,2 & 3 for the (1+32) and (1+128)-CGP over a budget of over more than 10000 fitness evaluations. It can be seen that the diversity of both, the (1+32) and (1+128)-CGP, is below 100% over the budget of fitness evaluations. We will analyze this finding in Subsection 9.3.3.

9.3.2 Point Mutation vs. Random Initialization

The results of the previous subsection indicate that the majority of the fitness evaluations of algorithms with medium size and big populations does not contribute to the search performance of CGP in the Boolean function domain. To analyze this hypothetical assumption in more detail, we determined the frequency of fitness values which have been achieved after a predefined number of point mutations on the genotype and the fitness values of random initialization of the genotype. The fitness functions of all tested problems are minimizing ones. We analyzed the fitness space by measuring the distribution of the fitness values in the search space, which has been achieved by point mutations and random initializations. We performed 10^7 random genotype initializations and point mutations. Afterward, we calculated the frequency for each achieved fitness value. We determined the fitness values of point mutations and random initializations for every tested problem from the previous subsection. We determined these so-called fitness histograms for all benchmark problems in the Boolean function and the symbolic regression domain.
Figure 9.7 shows the results for the comparison between random initialization and point mutation for the Boolean function problems in the Boolean domain. It can be seen that the distributions of random initialization and point mutation are similar and that the majority of the achieved fitness values are located in the same range for every tested problem. However, it can also be seen that the histograms are not exactly equal and that each histogram on the point mutation site has a peak in

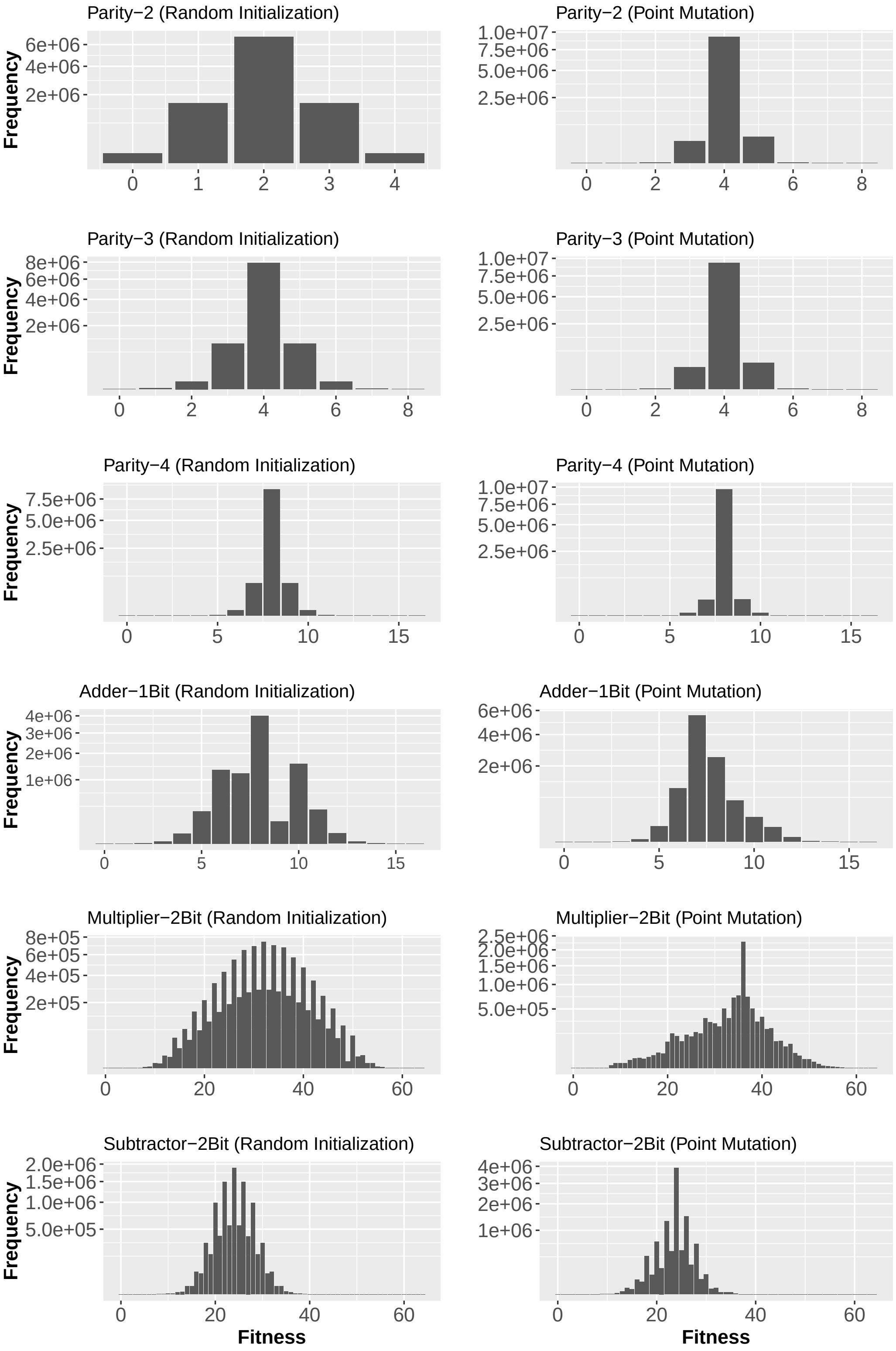


Figure 9.7: Histograms of the comparison between random initialization and point mutations for the tested Boolean function problems.

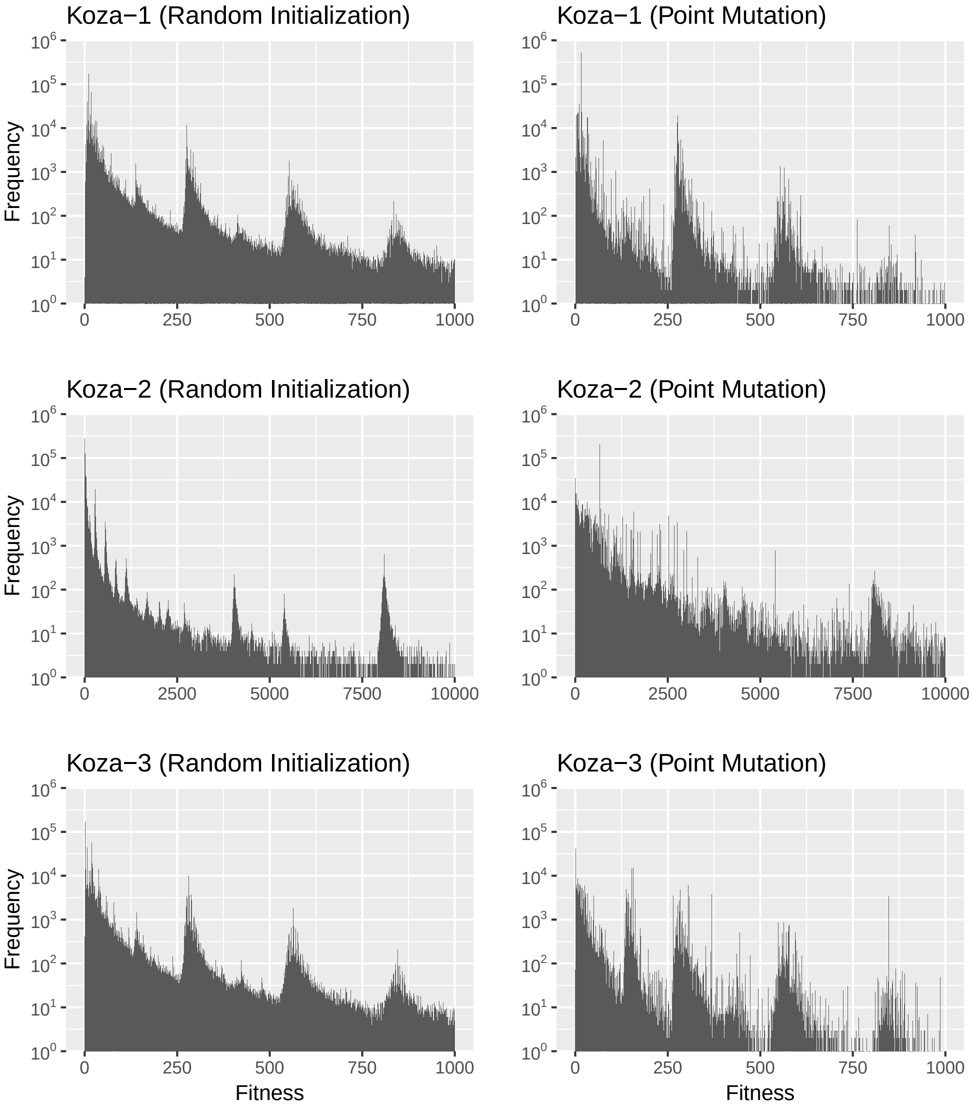


Figure 9.8: Histograms of the comparison between random initialization and point mutations for the tested symbolic regression problems.

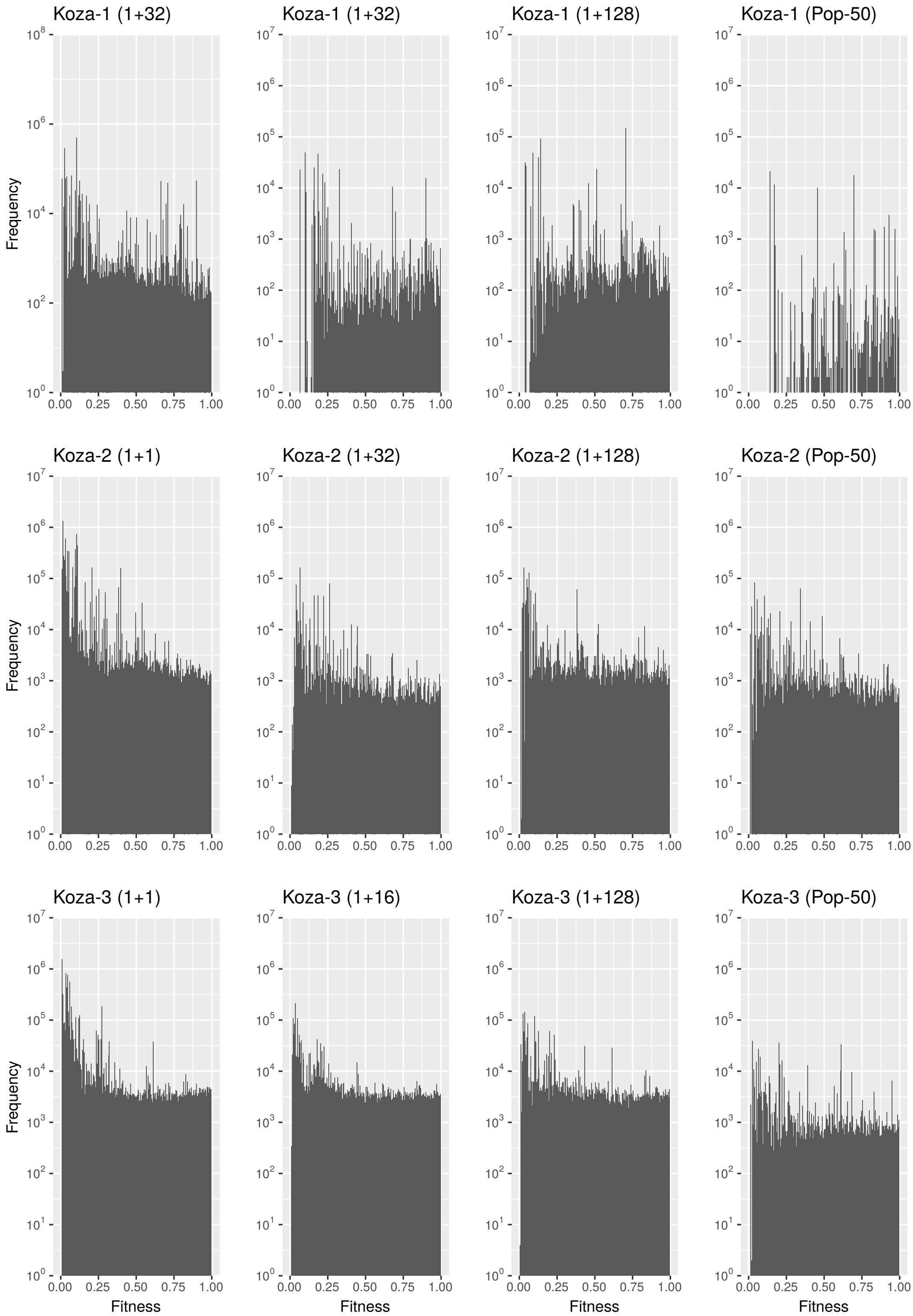


Figure 9.9: Histograms of the comparison between random initialization and point mutation for the tested symbolic regression problems in the area near the global optimum.

which the frequency of a certain fitness value is up to one tens potency higher when compared to the maximum frequency on the random initialization side. An explanation for these peaks could be the so-called *silent mutations*, which don't cause a change of the fitness value. Consequently, we may achieve another distribution of the frequency of fitness values. However, since it is visible that the obtained fitness values are located in the same range for random initialization and point mutation, we use our findings for the following analysis of the respective research question. Figure 9.8 shows the results for the symbolic regression problems. It is also clearly visible that the distributions of the fitness values of the random initialization and point mutation are also similar. However, it can be seen that the distributions of the Boolean function problems and the symbolic regression problems differ markedly from each other. The distributions of the tested Boolean function problems seem to be centralized in the space of possible fitness values, which is contrary to the distributions of the symbolic regression problems. On the three tested symbolic regression problems, the distributions are directed toward the global optimum and the most frequent fitness values are located close to the global optimum. On the basis of this finding and related to the observations of our analytic comparison between the $(1 + \lambda)$-CGP and the Pop-50-CGP in Section 9.3, we determined the frequency of occurrence of fitness values which are located close to the global optimum. More precisely, we measured the frequency in the area where the cost function value is greater than 0.01 but less or equal 1.0 . Following the experiments in Section 9.3 with the a setting of $\lambda \in \{1, 16, 32{,}128\}$, we determined the fitness histograms for these settings of the λ parameter and the Pop-50-CGP. The histograms are shown in Figure 9.9. In comparison to the results when higher values for λ parameter are used or if the Pop-50-CGP is used, the $(1 + 1)$-CGP shows a higher frequency of fitness values very close to the global optimum. Moreover, the peaks which can be spotted in the histogram are up to one power of ten higher for the $(1 + 1)$-CGP.
We will discuss the significance of these observations in the analysis of the research questions.

9.3.3 Discussion and Analysis of Research Questions

The results of our experiments in this section shed more light on the behavior of two algorithmic approaches to the use of CGP in two different problem domains. Our results indicate that the exploration behavior of random initialization and point mutation is similar. This finding, which we determined with the help of fitness histograms, is an important one to explain the contrary situation in the Boolean and symbolic regression domain. The determined distributions of Boolean function problems show that the majority of the fitness values, which were calculated after the mutations of the genotype, are located in a nearly central area of the fitness space. In the first place, this finding explains why the $(1 + 1)$-CGP algorithm performed most active on all of our tested problems. When the parent is mutated, it is likely that the fitness value of the offspring is located in the centralized area. In this way, the distribution for all tested problems can be described as unimodal.

Consequently, our experiments indicate that if a bigger population is used for our tested problems, it is also very likely that a certain amount of genetic variation and fitness evaluation steps are in a way *wasted*. This is because there is a high probability that the resulting fitness values are located in the high frequent central area of the space of fitness values. Consequently, our search performance evaluations for the tested Boolean problems indicate that the majority of the genetic variation and fitness evaluations steps do not contribute to the search performance of CGP because the fitness value of the offspring is not better as the fitness value of the parent.The $(1+1)$-CGP breeds and evaluates only one offspring per generation, and if the fitness value of the offspring is better, the offspring replaces the parent. Our analysis indicates that the number of wasted fitness evaluations of the $(1+1)$-CGP is lower in comparison to the other tested population sizes.

In the symbolic regression domain, our fitness histograms show entirely different distributions of the fitness values. The fitness values are more spread over the fitness value space, and the density is much higher compared to the Boolean function problems. Furthermore, the analysis of the search space with fitness histograms revealed high frequent fitness values near the global optimum. The existence of these high frequent fitness values indicates the existence of local optima. We also determined fitness histograms in the region close to the global optimum for $(1+\lambda)$-CGP and the Pop-50-CGP and observed higher frequencies in this region for the $(1+1)$-CGP. Since the search performance of the $(1+1)$-CGP on the three symbolic regression problems was inferior when compared to the results with other λ settings, a correlation between the existence of local optima and search performance can be drawn.

The distribution of the fitness values of the three tested symbolic regression problems can be considered as multimodal. Our determined fitness histograms of the symbolic regression problems make it clear with which situation the respective CGP algorithm has to deal with. On the one hand, the fitness value space is much denser and bigger, and our analysis indicates that this situation makes significant demands on the exploration abilities of the respective algorithm. Moreover, since the fitness histograms revealed many local optima, the respective CGP algorithm also has to challenge with this circumstance. We observed the best search performance on all three tested regression problems when the Pop-50 algorithm was in use. We also found a lower median of fitness evaluations as the $(1+1)$-CGP for the $(1+\lambda)$-CGP with higher settings of the λ parameter. Notably, a setting of $\lambda = 16$ and 32 led to good results on the tested problems. To find answers why these two configurations of the λ parameter and the Pop-50 algorithm showed better results as the $(1+1)$-CGP and the $(1+128)$-CGP, we investigated the convergence behavior for these algorithms. On all three regression problems, we observed the steepest convergence curves for the Pop-50 CGP. The $(1+1)$-CGP overall shows curves that are much flatter than the curve of the Pop-50 CGP, especially for the later generations. In the first place, this indicates that the Pop-50 CGP escapes local optima faster than the $(1+1$-CGP. Another indicator of this assumption is the observation of our fitness histograms. In all histograms, it is visible that the most frequent fitness values are close to the global optimum. Therefore, we reason that the CGP algorithms have to tackle these top frequent local optima in the later stage of the evolutionary process on the tested problems.

Another interesting observation are the curves of the $(1+\lambda)$ algorithm with $\lambda = 16, 32$ and 128. For the later generations, the curves are mainly located between the curves of the Pop-50 CGP and the $(1+1)$-CGP. In the front and middle section of the evolutionary process, the curves of the $(1+1)$-CGP are steeper than the curves of the $(1+\lambda)$ algorithm with $\lambda = 16, 32$ and 128. One question which turns out is why the $(1+128)$-CGP is inferior to the $(1+16)$ or $(1+32)$-CGP. One indicator can be found in the convergence plots. The convergence speed of the best individuals with $\lambda = 16, 32$ and 128 can be classified in the same class when compared to the convergence speed of the $(1+1)$-CGP and the Pop-50 CGP. However, we reason that the $(1+128)$-CGP causes more runs, which took a huge number of fitness evaluations to converge. A significant indicator for this reason is the third quartile in Table 9.19. Our reason is also backed by another indicator which we determined by the measurement of the diversity for the three tested regression problems, shown in Figure 9.6. The results of our measurements for the $(1+32)$-CGP and $(1+128)$-CGP demonstrate that there is a certain amount of individuals in the population which are equal to the parent. Furthermore, we assume that these same-fit individuals are caused by mutations that only flip inactive genes and have no effect on the phenotype. Consequently, the fitness evaluation may be wasted because it seems that these individuals do not contribute to the search performance.

Research Question. (Wasted fitness evaluations) *Does the use of middle-size and bigger populations in CGP lead to wasted fitness evaluations?*

Our experiments with the $(1+\lambda)$-CGP algorithm demonstrated the effectiveness of very small population sizes for our tested problems in the Boolean domains. Our first experiment indicates a correlation between the population size and the search performance for our tested Boolean function problems. This experiment included a measurement of the search performance with an exponential decrease of the λ parameter. Our analysis with fitness histograms indicate that for all tested Boolean function problems that the distribution of the fitness values makes the use of medium and big population sizes ineffective. The reason for this is that it is very likely that the fitness of the majority of the offspring which is bred from the parent per generation, just lay in a frequent high area within the space of fitness values. This centralized distribution has been observed on all tested problems.

In the symbolic regression domain, we observed a different situation. On all three regression problems, we observed a distribution that seems to be suitable for the use of a bigger population size as in the Boolean domain. The most frequent fitness values are located in the near of the global optimum. Furthermore, the distributions are more spread and have a much higher density. The presented experiments indicate that a population-based algorithm explores these spaces with a higher probability of an improvement of the fitness values.

Research Question. (Random Initialization vs. Point Mutation) *Cause random initialization and point mutation similar effects?*

For all of our tested problems in both problem domains, we observed similar effects when random initialization and point mutation were used. The observed effects are not equal because it appears that a large number of mutations do not lead to any change of the fitness value. These so-called *silent* or *neutral* mutations are well-known in the field of CGP. However, for the Boolean problems, it is evident that the most frequent fitness values are located in the same range. In the symbolic regression domain, it is also clearly visible that the shape is also very similar for all three regression problems. Moreover, we determined a unimodal distribution on the tested boolean function problem and a multimodal distribution on the symbolic regression problems with the use of random initialization and point mutation.

9.4 A Case Study on a Toy Problem

In this section, we analyze a typical toy problem in the symbolic regression domain. Since these types of problems have no practical relevance, some of these problems can be very useful to investigate GP algorithms in detail. For our case study we chose the very simple regression function $f(x) = x^2+x$. The purpose of this case study is to achieve more understanding and insight into the working mechanism of the subgraph crossover and its corresponding beneficial effects to the search performance. This section intends to shed more light into the question in which way the subgraph crossover contributes to the search performance.

9.4.1 Exploration Analysis in Phenotype Space

For our case study, we analyzed the exploration behavior in phenotype space on the chosen symbolic regression problem. This type of analysis is a simple way to analyze the exploration abilities of a CGP algorithm and is done by determining the number of different phenotypes that have been observed in a single run. To measure the number of different phenotypes in the population, we first determine the textual representation of each phenotype. For instance, in the symbolic regression domain, we can build a textual representation of the functionality in reverse polish notation. The determination of the textual representation can be done by the backward search, which is performed in the framework of the evaluation process. Each determined textual representation is stored in the hashmap. When the representation is

Table 9.20: Configuration of the $(1+4)$-CGP and Canonical-CGP for the node evaluation analysis experiments

Property	**$(1+4)$-CGP**	**Canonical-CGP**
Maximum node count	5	5
Number of inputs	2	2
Number of outputs	1	1
Population size	5	50
Function set	$+, -, *, /$	$+, -, *, /$
Mutation rate	0.1	0.1
Crossover rate	-	1.0
Tournament selection size	-	4
Elitism size	-	2

Table 9.21: Results for the comparison between $(1+4)$-CGP and Canonical-CGP on the problem $f(x) = x^2 + x$ evaluated by the number of fitness evaluation to termination

Problem	**Algorithm**	**Mean FE**	**SD**	**SEM**	**1Q**	**Median**	**3Q**
$f(x) = x^2 + x$	$(1+4)$-CGP	905	1085	±108	241	588	1062
	Canonical-CGP	**401**‡	**726**	**±72**	**48**	**72**	**336**

already stored in the hashmap, we increase the frequency counter of this textual representation. In this way, we can easily measure how many different phenotypes have been discovered after a certain number of fitness evaluations.
The CGP has been equipped with a small number of function nodes. We investigated the search abilities for the $(1+4)$-CGP and the Pop-50 algorithm and compared the results of the exploration analysis. The algorithm configuration for both algorithms is shown in Table 9.20. For the exploration analysis in the phenotype space, we investigated different budgets of fitness evaluations (48, 96, 144, 192, 240). For each budget, we performed 100 runs and measured the size of the hashmap at the end of each run.

Table 9.21 shows the results of the comparison for the symbolic regression problem, and it is seen that the Canonical-CGP with crossover performs significantly better compared to the $(1 + 4)$-CGP. Figure 9.10 and show the results of the exploration analysis in the phenotype space, and it is seen that the exploration level of the Pop-50 algorithm is much higher for the budget of 48, 96 and 144 fitness evaluations compared to the $(1 + 4)$-CGP. However, for the budget of 192 and 240 fitness evaluations, it is also visible that the exploration stagnates, but it can also be seen in Table 9.21 that the Pop-50 algorithm finds the ideal solution with a much smaller amount of fitness evaluations when compared to the $(1 + 4)$-CGP.

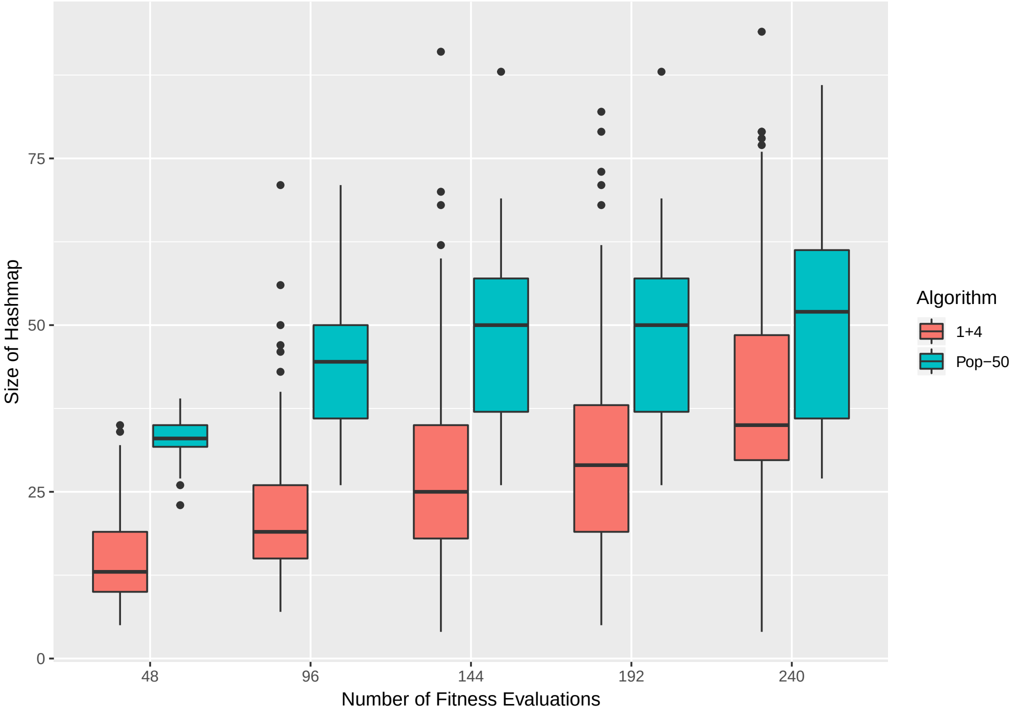


Figure 9.10: Boxplots for the exploration analysis in phenotype space

9.4.2 Structural Phenotype Analysis

To investigate the working mechanism and effects of the subgraph crossover in detail, we performed an analysis of the phenotypes itself. On the one hand, we measured the distance between a particular phenotype that belongs to an individual and the ideal solution. We also measured the number of ideal solutions which have been found just after a crossover operation. The goal of this analysis was to investigate in which way the use of the subgraph crossover influences the evolutionary run on the phenotypic level. The experimental setup for this analysis is very similar to the exploration analysis in the previous subsection. For our experiments, we used Canonical-CGP with different crossover rates. We investigated the Canonical-CGP with 25%, 50%, 75% and 100% of crossover. The algorithm configuration is shown in Table 9.22. To maintain the simplicity of the given task for the CGP, we only used the small Koza function set, which includes four basic arithmetic functions.

We measured the search performance of different rates of crossover. The search performance was determined with the *fitness-evaluations-to-termination* method. The results of our comparison are shown in Table 9.23, and as visible, the increase of the crossover rate decreased the average number of fitness evaluations which were necessary to find the ideal solution. Based on this finding, we investigated the effect of different rates of crossover in phenotypic space. More precisely, we measured the distance of the textual representation of a parent and an offspring to

Table 9.22: Configuration of the Canonical-CGP for the phenotypic analysis

Property	**Setting**
Maximum node count	5
Number of inputs	2
Number of outputs	1
Population size	50
Function set	$+, -, *, /$
Mutation rate	0.1
Crossover rates	0.25/0.5/0.75/1.0
Tournament selection size	4
Elitism size	2

Table 9.23: Results for the problem $f(x) = x^2 + x$ evaluated by the number of fitness evaluation to termination

Problem	**Crossover rate**	**Mean FE**	**SD**	**SEM**	**1Q**	**Median**	**3Q**
$f(x) = x^2 + x$	0 %	925	1330	±133	48	240	1212
	25%	595	861	±86	48	144	864
	50%	531	929	±92	48	144	864
	75%	531‡	914	±91	48	72	540
	100%	**401**‡	**726**	±**72**	**48**	**72**	**336**

the textual representation of one ideal solution T_I . One of the most simple textual representations of the ideal solution is `+ x0 (* x0 x0)` in which `x0` represents the first input of the cartesian program. We measured the Levenshtein distance (LD) [76] to this textual representation of the ideal solution before and after the crossover procedure. To determine the LD of the first parent and the offspring, we generated the textual representation of the parent and the offspring. More precisely, we first dertermined the LD between the textual representation of the first parent T_{P1} and the ideal solution T_I which is denoted as D_{PI}. After the crossover procedure we measured the LD between the textual representation of the offspring T_O and the ideal solution which is denoted as D_{OI}. To determine the difference between these two LD measurements we subtracted D_{PI} from D_{OI}. We performed 100 runs in total and summarized the differences which were calculated after each crossover operation over all runs. To determine the mean, we divided the sum of the differences by the total number of differences, which were determined for the respective crossover rate. Please note that a negative difference of the LD measurements means that the textual representation of the offspring is more similar to the textual representation of the ideal solution after the crossover procedure. Besides the mean difference, we calculated the standard deviation, minimum, and the maximum difference.

Table 9.23 shows the results of the search performance evaluation and Table 9.24 shows the results for the Levenshtein distance analysis. It is visible that the increase of the crossover rate leads to a gradual decrease in the total difference of the Levenshtein distance. Moreover, we observed a negative difference in the Levenshtein

Table 9.24: Results of the Levenshtein distance (LD) analysis for the problem $f(x) = x^2 + x$

Problem	Crossover rate	Number of measurements	Total LD difference	Mean LD difference	SD	Min diff.	Max diff.
$f(x) = x^2 + x$	0 %	–	–	–	–	–	–
	25%	1578	−560	−0.35	4.54	−36	34
	50%	3352	−1129	−0.34	4.72	−69	55
	75%	4242	−1302	−0.31	4.95	−53	55
	100%	5284	−1633	−0.31	5.06	−40	42

Table 9.25: Results for the experiment to determine the hit - crossover relationship

Problem	Crossover rate	Number of Hits
$f(x) = x^2 + x$	0 %	–
	25%	3
	50%	8
	75%	11
	100%	20

distance for all crossover rates.

9.4.3 Analysis of the *Hit* - Crossover Relationship

Our experiments in the previous subsection unveiled one effect, which is caused by the use of the subgraph crossover. However, our experiments do not answer the question of why higher rates of crossover contribute significantly to the search performance. To achieve a more detailed view on the influence of the respective crossover rate, we measured the number of *Hits*, which occurred directly after a crossover operation. We increased the number of *Hits* by one when the offspring matches the ideal solution. The algorithm configuration is the same as in the previous subsection. We performed 100 runs in total and evaluated each run until the ideal solution was found. Intending to analyze the *Hits* in detail, we generated the textual representation in reverse polish notation of both parents and the offspring. We present the textual representations of various *Hits* in special listings.

Table 9.25 shows the results for the hit-crossover relationship determination, and it is visible that the increase of the crossover rate leads to a gradual increase of the number hits after a crossover operation. Figure 9.11 shows various examples of *Hits*. It can be seen that parts of one parent are integrated into the phenotype of the other parent, which directly leads to the determination of the final solution. We will discuss and analyze this finding in the next section.

9.4.4 Discussion

The results of the exploration analysis in phenotype space indicate that the use of the Pop-50 CGP in combination with the subgraph crossover results in better

```
Parent  1:    + x0 (- x0 x0)
Parent  2:    * x0 x0
Offspring:    + x0 (* x0 x0)
```

Example 1

```
Parent  1:    + x0 (/ x0 x1)
Parent  2:    * x0 (/ (* x0 x0) x1)
Offspring:    + x0 (* x0 x0)
```

Example 2

```
Parent  1:    * x0 x0
Parent  2:    + (- x0 x0) x0
Offspring:    + (* x0 x0) x0
```

Example 3

```
Parent  1:    * x0 (* x1 x1)
Parent  2:    / x0 (+ x1 x0)
Offspring:    * x0 (+ x1 x0)
```

Example 4

```
Parent  1:  * x0 x0
Parent  2:  + (/ (- x0 x0)(- x0 x0)) x0
Offspring:  + (* x0 x0) x0
```

Example 5

```
Parent  1:    * x1 x0
Parent  2:    + (* x0 (+ x0 x1)) x0
Offspring:    + (* x0 (* x1 x0)) x0
```

Example 6

```
Parent  1:    * (- x1 x1) (+ x1 x0)
Parent  2:    * (* x1 x0) x1
Offspring:    * (* x1 x0) (+ x1 x0)
```

Example 7

```
Parent  1:    * x0 x1
Parent  2:    * x0 (+ (* x0 x0) x1)
Offspring:    * x0 (+ (* x0 x1) x1)
```

Example 8

Figure 9.11: Various examples of successful crossover operations

exploration of the phenotype space on our tested symbolic regression problem. Furthermore, the results of our search performance showed that the Pop-50 algorithm is superior to the traditional (1 + 4)-CGP on that problem. One purpose of this section was to investigate the effects of phenotype space, which are caused by both algorithms. Even if we observed an increased exploration of the phenotype space and a reduced number of fitness evaluations until the ideal solution was found, it is too early to make any general claims. We must emphasize that we found a trend, and this finding is based on a case study which investigated only one symbolic regression problem. To make general claims and more significant statements, a more rigorous and more comprehensive study has to be performed.

The results of our structural analysis unveiled two exciting effects of the subgraph crossover for the given problem in phenotype space. In the first place, our experiments showed that the use of the subgraph crossover decreases the Levenshtein distance when the textual representation of the first parent and the offspring are compared to the ideal solution. Moreover, this type of analysis also indicates that there exists a correlation between the crossover rate and the mean difference of the Levenshtein distance to the ideal solution.

Our experiments also showed that the use of the subgraph crossover increases the chance that the ideal solution is found after a crossover operation for the given toy problem. However, these findings are also based on our case study with only one regression problem, and for more significant statements, we have to perform experiments with a large number of the issues. This also means that other benchmarks

of other problem domains have to be analyzed. The reason for this is that we have to investigate if the effects also occur on different types of problems. A more comprehensive study could also answer the question if the two effects which have been observed in our experiments correlate with an improvement of the search performance of the Pop-50 CGP. However, we investigated the occurrence of *Hits* after a crossover operation for the given toy problem. The presented examples clearly show that small phenotypes are integrated into bigger ones. Moreover, the fitness of both phenotypes must be on a certain fitness level since the tournament selection sorts out individuals with low fitness.
An answer to the question in which way the subgraph crossover contributes to the evolutionary search of CGP, might be found on the basis of the so-called *Building Block Hypothesis* (BBH), which has been formulated as an explanation of GAs by Goldberg [31], According to Goldberg:

> “Short, low order, and highly fit schemata are sampled, recombined, and resampled to form strings of potentially higher fitness. ”

Goldberg [31, p. 41]

Especially in the first three examples of the listings in Figure 9.11, it can be seen that the short schemata `* x0 x0` is recombined into a longer schemata which finally leads to the determination of the ideal solution. However, for more significant statements about the BBH as an possible explanation for the improved search behavior when subgraph crossover is used, more comprehensive studies have to follow. It should be also mentioned that the BBH received skepticism concerning weak theoretical foundations, incoherence and.

9.5 Redundancy and Fitness Space Analysis

Former experiments in the field CGP with boolean function problems showed that the increase of the genotype length improves the search performance. However, this finding in the boolean domain has been generalized throughout CGP history. In this section, we analyze why this performance dogma does not hold for three symbolic regression benchmark problems that we already evaluated in Section 9.2. Furthermore, our experiments give answers to why the search performance does not improve by increasing the genotype length on these problems. This section also demonstrates the role of redundancy for the evolutionary search in discrete and bounded fitness spaces.

We analyzed the search performance on Koza 1, 2 & 3 problem. We first measured the search performance of all symbolic regression problems with genotype lengths of 10, 20, 50, and 100 function nodes. The algorithm configuration for the experiments is shown in Table 9.26. We performed 100 runs for each experiment and measured the search performance by the number of generations until the ideal solution was found. In addition to the mean values of the measurements, we calculated the standard

Table 9.26: Configuration of the $(1+4)$-CGP

Property	**$(1+4)$-CGP**
Maximum node count	10/20/50/100
Mutation rate [%]	20/10/8/6
Number of inputs	2
Number of outputs	1
Population size	5
Function set	$+, -, *, /$

Table 9.27: Number of fitness values for various genotype length for the problems Koza 1,2 & 3

Problem	**Number of function nodes**	**Number of fitness values**
Koza-1	10	15655
	20	53695
	50	172244
	100	306242
Koza-2	10	15646
	20	54133
	50	173122
	100	307665
Koza-3	10	15859
	20	54823
	50	173523
	100	307412

deviation (SD) and the standard error of the mean (SEM). We used the $(1+4)$-CGP algorithm and investigated the search performance of the algorithm with the standard *continuously* fitness function which has been used in Section 9.2. When this fitness function is used, the fitness values can vary in a range $\mathbb{R}_{>=0}$. Since the range of this fitness function is not fixed, we evaluated the size of the fitness space for genotype lengths with 10, 20, 50 and 100 function nodes by sampling 10^6 random genotypes. We evaluated the fitness of each genotype and stored its frequency in a hash map. Afterward, we used the size of the hash map to conclude the magnitude of the space of fitness values. Intending to demonstrate the role of redundancy for the search performance of CGP, we investigated all benchmark functions with discrete fitness. In these types of experiments, the fitness function was similar to the *continuously* fitness function, with the exception that the fitness values were discretized within a range of whole numbers from 0 to 20. In this way, the experiments covered the investigation of the search performance for different lengths of the genotype on three regression problems evaluated with continuous as well as discrete fitness. The algorithm configuration is shown in Table 9.26. The mutation rates have been determined empirically and are oriented with results of former experiments of this chapter.

Table 9.28 shows and Figure 9.12 illustrates the stylized search performance behav-

Table 9.28: Results for the symbolic regression problems Koza 1,2,3 when continuous fitness is used

Problem	Number of function nodes	Mean Fitness Evaluations	SD	SEM	1Q	Median	3Q
Koza-1	10	456547	749589	±74958	10889	47852	363278
	20	536685	814927	±81493	11019	56704	584147
	50	1373228	662682	±66268	815347	1586900	2000000
	100	1407718	709660	±70966	698582	1934054	2000000
Koza-2	10	466726	667196	±66719	27233	123156	533127
	20	695164	768904	±76890	51434	281468	1381271
	50	808779	800203	±80020	56950	525106	1830031
	100	942653	877873	±87787	96501	709382	2000000
Koza-3	10	638451	756151	±75615	52473	259490	1097556
	20	774234	812167	±81216	43923	476620	1842339
	50	867915	826392	±82639	67928	540420	2000000
	100	1154141	902271	±90227	145972	1852700	2000000

Table 9.29: Results for the symbolic regression problems Koza 1,2,3 when discrete fitness is used

Problem	Number of function nodes	Mean fitness evaluations	SD	SEM	1Q	Median	3Q
Koza-1	10	3694	3300	±330	1116	2840	4931
	20	3346	3870	±387	1051	2268	3925
	50	3300	4165	±416	764	2114	4183
	100	2798	3812	±381	609	1470	3173
Koza-2	10	8589	8768	±876	2062	5984	11468
	20	5478	5518	±552	1650	3880	7973
	50	3076	5154	±515	684	1414	3226
	100	1998	2469	±246	548	1184	2178
Koza-3	10	2500	2852	±285	617	1808	3435
	20	2011	2238	±223	487	1156	2860
	50	1212	1220	±122	315	846	1719
	100	1145	1107	±110	283	832	1639

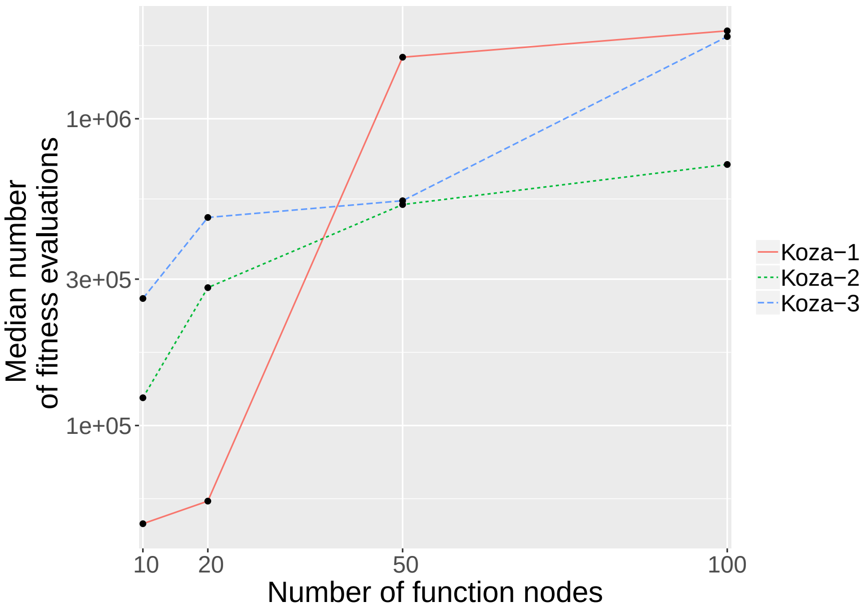


Figure 9.12: Stylized search performance behavior for various genotype lenghts on continuous fitness

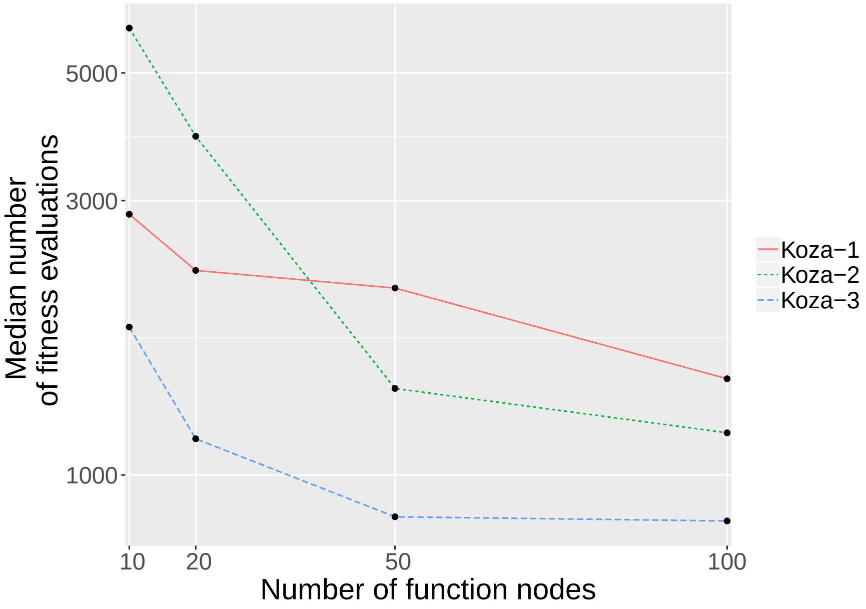


Figure 9.13: Stylized search performance behavior for various genotype lenghts on discrete fitness

ior for various genotype lengths when continuous fitness is used. It is visible that when the increase of the length of the genotype, the search performance decreases. Table 9.27 shows the results of the analysis of the fitness space for the three Koza problems. It can be seen that the size of the fitness space increases when the length of the genotype is increased. Table 9.29 shows and Figure 9.13 illustrates the search performance behavior for various genotype lengths when discrete fitness is used. Here, it is visible that increasing the length of the genotype increases the search performance.

9.5.1 Analysis of the results

The results of our experiments clearly show that the correlation between the improvement of the search performance and the increase of the genotype length can not be generalized. On all three regression functions, the rise in the genotype length increased the size of the space of fitness values rapidly. This finding indicates that the level of redundancy decreases by increasing the genotype lengths for all three tested regression functions. Our experiments also showed that the increase of the genotype length also deteriorated the search performance on all three tested problems. Moreover, our results also indicate a correlation between the size of the space of fitness values and the search performance of the $(1 + \lambda)$-CGP algorithm. In this way, our results confirm the findings of Miller et al., who investigated the role of redundancy and its meaning for the computational efficiency of CGP when discrete fitness is in use. However, since Miller experiments mostly focused on boolean function problems for which the respective candidate solutions are evaluated with discrete fitness. Moreover, the fitness value space of the most popular boolean function benchmark problems is fixed and cannot vary when the parametrization is changed. In this way, we can conclude that our results contribute to the state of knowledge about the correlation between the type of fitness and computational efficiency in CGP.

10 Summary and Outlook

10.1 Analysis of Hypotheses

Hypothesis 1 (Population size). *Small populations perform most effective in CGP.*

One of the most significant key publications for this claim is the work of Miller [97]. The results of Miller's experiments showed that Koza's *Computational Effort* is smaller if small population sizes are in use. However, the experiments focused only on Boolean function problems. Another key publication for the use of small population sizes in CGP is the work of Miller and Smith [99]. For a clearer analysis of this claim, the impact of the population size on the performance of CGP was comprehensively investigated in Chapter 5 and Chapter 9. It has been demonstrated that the claim of small population sizes cannot be generalized. However, in the Boolean domain, the results and findings of former studies seem to be coherent but it has been demonstrated that this claim does not hold for the symbolic regression. The comparative studies in Chapter 5 and Chapter 9 clearly show that medium and high population sizes perform significantly better than the traditional $(1+4)$-CGP in the symbolic regression domain. In Section 9.3 of Chapter 9, we investigated the search performance and convergence behavior of small, middle-sized, and big population sizes on well-known benchmark problems. For the boolean function problems, we observed that a very small population size performs best on these problems. However, for the tested symbolic regression problems, we observed that the middle-sized population can lead to a better search performance. We also found reasons and indicators in the analytic part of Chapter 9, which explain our observations.

Hypothesis 2 $(1 + \lambda$-CGP $)$. *The* $1 + \lambda$*-CGP algorithm is the most effective way to use CGP.*

The key publication of this claim is Miller's [97] empirical study in CGP, in which he stated that recombination doesn't seem to add anything to the search performance. Another work by Miller and Thompson [101], in which CGP itself was introduced, questioned the benefits of crossover in CGP. The results of both works led to the predominant use of the popular $(1+\lambda)$-CGP. For a reevaluation of this claim, we presented comprehensive experiments in Chapter 5 and Chapter 9. The results of both studies show that the $(1 + \lambda)$-CGP is not the predominant evolutionary algorithm for the use of CGP. In our experiments, the $(\mu + \lambda)$-CGP and the Canonical-CGP were used with crossover. The results clearly show that both algorithms perform

significantly better on a wide range of different problems.
In Section 9.3 of Chapter 9, we investigated the search performance and convergence behavior of the $(1+\lambda)$-CGP and the Pop-50 CGP. On the boolean function problems, the $(1+\lambda)$-CGP showed the best results and was used with a very small population size. On the three tested symbolic regression problems, the tested Pop-50 CGP with crossover clearly outperformed the $(1+\lambda)$-CGP algorithm.

Hypothesis 3 (Redundancy). *Extremely large genotypes perform most effectively in CGP.*

For the analysis of hypothesis 3, we investigated the search performance of CGP after automatic parameter tuning in Chapter 5 and Chapter 9. The claim that extremely high levels of redundancy, caused by extremely large genotypes, perform most efficient in CGP grew out of Miller and Smith's work [99]. Their experiments on Boolean function problems showed that CGP performed best with extremely high levels of redundancy on the Even-Parity-3 problem and the 2-Bit digital multiplier problem. The experiments were performed using the standard $(1+\lambda)$-CGP algorithm and a small population size of five individuals.

The results of the experiments in Chapter 5 and Chapter 9 indicate that the efficiency of the combination of high levels of redundancy and small population sizes is valid for the majority of the tested Boolean function problems. However, on most tested symbolic regression problems, smaller genotypes and bigger population sizes performed best. The outcomes of our comparative studies show that the dogma of high levels of redundancy cannot be generalized. Furthermore, in Chapter 7 and Chapter 8 it has been shown that also small genotypes can be efficiently used and recombined in CGP. These findings were investigated in more detail on three popular regression functions in Chapter 9, Section 9.5. The results of the experiments indicate that Miller and Smith's findings hold for discrete fitness spaces with fixed boundaries but can't be generalized for other types of fitness such as continuous fitness. Moreover, the experiments in this section also found an indicator that could give an answer to why the stated redundancy dogma does not hold for continuous fitness spaces. An analysis of the size of the search space for all three tested regression functions showed that the increase of the genotype increases the size of the fitness value space. The results of the search performance experiments with continuous fitness indicate that the $(1+4)$-CGP algorithm is not an effective choice for the evolutionary search in these large fitness spaces. In Chapter 9, Section 9.3, it has been demonstrated that other algorithmic approaches with crossover can be more effective when large fitness spaces must be explored.

Chapter 9, Section 9.5 investigated the role of redundancy in continuous and discrete fitness spaces on three symbolic regression problems. The investigation started with the determination of the search performance within a continuous fitness space. The experiments were repeated within a discretized search space afterward. The outcome of this study was that we demonstrated that small genotypes can be effective within continuous fitness spaces. After the discretization of the fitness space, we observed that larger genotypes improve the search performance on the tested problems. The discrete fitness spaces of popular boolean function problems are fixed and can't vary in size when the length of the genotype is changed. However, for the continuous fitness spaces of the regression problems Koza 1,2, and 3, we observed another situation: When the length of the genotype is increased, the size of fitness space also increases. Our observations on the search performance with continuous indicate that the increase of the search space decreases the search performance.

Hypothesis 4 (Crossover). *Crossover does not contribute to the search performance of integer-based standard CGP.*

The hypothesis that crossover does not contribute to the search performance is rooted in the work of Miller [97] and Miller and Thompson [101]. The experiments of both works led to a general question of the benefits of crossover in CGP. However, further work, which has been contributed to standard CGP, rejected the use of crossover and focused on the well-known $(1 + \lambda)$-CGP. The complete history of the role of crossover in the field of CGP was analyzed in Chapter 7. To shed more light on the question of crossover in CGP, two new crossover techniques were proposed in Chapter 7. The crossover techniques were evaluated on various benchmark problems, and its beneficial effects have been made clear in the experiment section of Chapter 7 and Chapter 9. Overall, our experiments demonstrate that crossover can contribute to the search performance of CGP.

In Section 9.3 of Chapter 9, we investigated the search performance and convergence behavior of the $(1+\lambda)$-CGP and canonical EA. The canonical EA was equipped with the subgraph crossover technique and outperformed the $(1 + \lambda)$-CGP on the tested regression problems Koza 1,2, and 3. A study of the distribution of the fitness values and the convergence behavior indicates that the Pop-50 CGP can overcome local optima faster than the $(1 + \lambda)$-CGP.

Hypothesis 5 (Mutation). *The standard CGP point mutation operator is sufficient for the mutative variation.*

Our experiments in Chapter 8 and Chapter 9 outlined the limitations of the sole use of the point mutation operator. Furthermore, we have investigated the limitations with active function node analysis. The results of the experiments with the *insertion* and *deletion* mutation techniques in Chapter 8 and Chapter 9 demonstrate that the mutative variation can be improved using advanced phenotypic mutations. More precisely, the experiments demonstrate that the use of these advanced mutation methods can result in an improved search performance on a diverse set of benchmark problems. Moreover, the presented active function node analysis in Chapter 8 indicates that the use of the *insertion* and *deletion* mutation enables a wider search in the phenotypic space.

Hypothesis 6 (Boolean function domain). *The* $(1 + \lambda)$*-CGP algorithm performs most effective in the Boolean domain.*

For the analysis of this hypothesis, we investigated the search performance of different CGP algorithms in Chapter 5. Our experiments show that the predominant role of the $(1+\lambda)$-CGP in the Boolean domain cannot be generalized. The design of the study included *state-of-the art* benchmarks in the Boolean domains such as the 3-Bit digital multiplier and high order parity-even problems, which are commonly used in the field of CGP. The results of our experiments show that the mutation-only $(\mu + \lambda)$-CGP algorithm is superior to the $(1 + \lambda)$-CGP on some Boolean problems. Furthermore, our results in Chapter 9 show that the predominant role of mutation-only CGP algorithms in the Boolean domain cannot be generalized. On most tested Boolean benchmarks, the recombination based $(\mu + \lambda)$-CGP performed best.

Hypothesis 7 (Symbolic regression domain). *The* $(1+\lambda)$*-CGP algorithm performs most effective in the symbolic regression domain.*

Similar to the analysis in the Boolean domain, we investigated the search performance of different CGP algorithms in Chapter 5. Our experiments show that the predominant role of the $(1 + \lambda)$-CGP in the symbolic regression domain cannot be generalized as well. The design of the study included *state-of-the art* benchmarks such as the Keijzer-6 and Pagie-1 problem, which have been proposed as benchmarks for the GP community. Throughout Chapter 9 we have demonstrated the potential of the subgraph and block crossover when it is used within a typical canonical GA fashion. Moreover, we also found indicators in the symbolic regression domain, which provide explanations for our assumptions that the Canonical-CGP with subgraph crossover explores the search space more effectively and escapes local optima more likely than the $(1 + \lambda)$-CGP.

10.2 Analysis of Research Questions

Research question 1 (Crossover). *Can some kind of crossover be effectively used in CGP?*

The experiments of our comparative studies in Chapter 7 and Chapter 9 demonstrate that the use of crossover can improve the search performance in different problem domains. The results in Chapter 7 and Chapter 9 demonstrate that the successful use of a crossover-based algorithm in CGP depends on a solid parameter tuning in which important parameters (e.g. crossover rate, population size, and tournament selection size) are adjusted to the given problem. A good example is the result of the arithmetic crossover in our comparative study. In the proposal of the arithmetic crossover, Clegg et al. [13] observed algorithm stagnation problems on a simple symbolic regression problem. In our comparative study, we tuned the parameters of the arithmetic crossover based real-valued CGP and on some of the tested problems, the algorithm outperformed the $(1+4)$-CGP. In Chapter 9 canonical EA equipped with the subgraph crossover showed overall solid results in the symbolic regression domain and outperformed the $(1+4)$-CGP on every tested problem. Moreover, the $(\mu+\lambda)$-EA equipped with the subgraph crossover outperformed the $(1+4)$-EA on all Boolean function problems. The use of the subgraph crossover also led to good results on both image operator design problems. Besides the subgraph crossover, we also evaluated the block crossover in Chapter 9 that showed promising results on several problems. However, compared to the subgraph crossover and in relation to the results of the $(1+\lambda)$-CGP, our results indicate that the subgraph crossover is a more stable and solid choice to use recombination in CGP.

Research question 2 (Theory). *How can CGP be analyzed on a theoretical level?*

The runtime analysis in Chapter 4 showed that *Multiplicative Drift Analysis* and *Artificial Fitness Levels* are suitable methods to analyze CGP on a theoretical level. With the use of both methods, we determined the upper and lower runtime bounds for two simple test problems.

Research question 3 (Real-valued CGP). *Has the real-valued CGP algorithm stagnation problems?*

We achieved some reasonable results for the real-valued CGP with arithmetic crossover in the search performance evaluation with the *best-fitness-of-run* method. These reasonable results were observed in the symbolic regression. and Boolean function domain. We determined important parameters with the help of automatic parameter tuning beforehand. However, the results were not better than the fitness values of the algorithm equipped with subgraph crossover. Moreover, in some cases, the real-valued CGP could not outperform the (1+4)-CGP. However, since former work has questioned the effectiveness of the arithmetic crossover, the results of our experiments underline the importance of parameter tuning in the field of CGP. Former work [13, 93] mainly focused on mere empirical evaluations, which led to early premature statements and recommendations. With the parameter settings, which we determined and used in Chapter 9, we did not observe excessive long run evolutionary runs as it has been observed by Clegg et al. [13] and Meier et al. [93]. Our experiments in Chapter 6 demonstrate that stagnation can occur under certain conditions (e.g. ineffective parametrization). Our experiments also indicate that the use of the presented adaptive strategy can contradict stagnation und these conditions. However, when the real-valued CGP is parameterized with effective settings, the contribution to the search performance of the adaptive strategy appears to be only marginally or in some cases destructive.

Research question 4 (Wasted fitness evaluations). *Does the use of middle-size and big populations in CGP lead to wasted fitness evaluations?*

This research question has been analyzed in Chapter 9, Section 9.3.

Research question 5 (Random Initialization vs. Point Mutation). *Cause random initialization and point mutation similar effects?*

This research question has been analyzed in Chapter 9, Section 9.3.

10.3 Recapitulation, Conclusion and Future Work

This thesis contributed to the fundamental knowledge of CGP and demonstrated opportunities for advanced genetic variation techniques in the field of CGP. In the first place, important and significant research questions were tackled, which have not been answered before the underlying work of this thesis was published. Moreover, this thesis investigated important scientific claims about the effectiveness and working mechanisms of CGP with the formulation and analysis of several hypotheses.

The most important contribution of this thesis to the field of CGP has been made with the proposal, evaluation and investigation of two new methods for crossover. Moreover, the results of comparative studies on crossover in Chapter 7 and Chapter 9 showed that a crossover-based algorithm can outperform the traditional and predominantly used $(1 + \lambda)$-CGP algorithm. This thesis also gave insight into the beneficial effects caused by the subgraph crossover on an experimental level.

The findings of the experiments in the symbolic regression domain also paved the way to disprove the generalization of the search performance dogma in CGP about the effectiveness of a small population and extremely high levels of redundancy. With the help of automated parameter tuning, the results of the performed comparative studies in Chapter 5, Chapter 7 and Chapter 9 unveiled a contrary situation between two problem major domains and disproved generalized statements about the parametrization of algorithms in CGP.

Another important contribution of this thesis is the first runtime analysis of CGP. The results of Chapter 4 showed how modern methods for the runtime analysis of evolutionary algorithms can be used to proof lower and upper bounds of simple test problems in CGP. This contribution also showed that for a theoretical analysis of CGP, different probabilities for different types of genes must be taken into account. This primarily relates to the function and connection genes for which different probabilities for successful mutations must be calculated.

In Chapter 8, the stagnation issues of the $(1+\lambda)$-CGP algorithm with the standard point mutation operator as the sole genetic operator, were analyzed and tackled. This has been accomplished by proposing two advanced phenotypic mutations for CGP. The analytic part of this work was devoted to a range analysis of the fitness values and an active function node analysis. On the one hand, these analyses unveiled disruptive effects of the point mutation operator. Alternatively, CGP showed a better exploration of the phenotype space when advanced mutation techniques are used. Another important part of this work is the comparison to Evolving Graphs by Graph Programming, which significantly outperformed CGP in the past on a set of popular boolean function problems. Moreover, on all tested benchmark problems, except the digital 3-Bit-Adder problem, a lower median number of fitness evaluations was achieved.

The evaluation in Chapter 9 demonstrated the effectiveness of crossover-based algorithm. Moreover, the analysis of a subgraph crossover-based algorithm and the traditional $(1+\lambda)$-CGP on three standard symbolic regression problems indicates that the tested subgraph crossover-based algorithm explores the phenotypic space better than the $(1+\lambda)$-CGP. The analysis of the $(1+\lambda)$-CGP also included a comparison of various settings of the λ parameter on three standard symbolic regression benchmarks. The comparison demonstrated that middle-size and big populations perform more effective on these problems. To analyze the reasons for these observations, we determined the distributions of the fitness values and determined a multimodal distribution. The histograms also indicate the existence of many local optima in the near of the global optimum. These types of analyses demonstrated that the use of a medium or a large setting of the λ parameter led to a lower frequency of these local optima. Furthermore, we observed the lowest frequencies of local optima with the Pop-50 algorithm equipped with subgraph crossover.

In the Boolean domain, a unimodal distribution of the tested problems was observed, and some of them were fairly symmetric. The search performance evaluation in the Boolean domain investigated three single-output and three multi-output problems. A setting of $\lambda = 1$ performed best on all tested problems, and these findings indicate a correlation between discrete unimodal fitness spaces and the effectiveness of very small population sizes.

With the help of the fitness value histograms, it was also possible to demonstrate the similarity of the effects caused by point mutation and random initialization. This chapter also gave answers to why big population sizes are ineffective on boolean and symbolic regression problems in CGP. A study on the distribution of the fitness values and the convergence behavior indicates that the Pop-50 CGP algorithm with crossover can overcome local optima faster than the traditional $(1 + \lambda)$-CGP. Another important point of this chapter was the investigation of continuous and discrete fitness spaces. The investigation on three symbolic regression problems with continuous fitness showed that by enlarging the genotype, the size of the fitness space enlarges too. This finding indicates why the corresponding search performance analysis showed the best results for the lowest setting of the genotype length.

However, for more general and significant statements, a large-scale study must follow in the future. More precisely, the analytic experiments must be expanded to shed more light on the search behavior of CGP in several problem domains. This can be seen as the most important point for future work, which follows up the results of this thesis. The reason for this is that the presented methods for analyzing CGP can significantly contribute to a detailed and comprehensive explanation of the working mechanisms and their corresponding effects. Since this thesis triggered a reconsideration of the predominant use of the $(1 + \lambda)$-CGP algorithm, other problem domains of CGP should be investigated in a similar way as it was done in Chapter 9. Concerning the effective use of crossover in CGP, it is a natural next step to expand the investigation to more problem domains. Another point which is left for future work concerns the implementation of the proposed methods. For our experiments we merely used naive Java implementations for the evolutionary computation research system ECJ. As a next step, the naive implementations should be improved with efficient implementations in the C/C++ programming language.

The following concluding remarks can be formulated on the basis of the outcome of this thesis:

- Phenotypic crossover and mutations operators can be used to improve the search performance of CGP
- The traditional use of CGP is outdated and has been comprehensively reconsidered
- Commonly used parametrization pattern can not be generalized in terms of effectiveness

- The conditions of the problem domain (e.g. size and structure of the fitness space) influence the parametrization and the sucesss of a certain EA in CGP
- The nature of the fitness function (i.e. discrete or continuous) plays a significant role for the search performance of a certain EA in CGP

Acronyms

ADF Automatically defined function. 25, 45

BBH Building Block Hypothesis. 210

CE Computational Effort. 17, 68

CGP Cartesian Genetic Programming. 15

EA Evolutionary Algorithm. 25, 29

EC Evolutionary Computation. 25

ECGP Embedded Cartesian Genetic Programming. 70

ECJ Java Evolutionary Computation Toolkit. 134

EGGP Evolving Graphs by Graph Programming. 16, 35, 166

ES Evolutionary Strategy. 31

FPGA Field-programmable gate array. 15, 46

GA Genetic Algorithm. 31

GAM Generalized Additive Model. 168

GI Genetic Improvement. 46

GP Genetic Programming. 15

GP2 Graph Programming Language 2. 35

GSGP Geometric Semantic Genetic Programmin. 45

HC Hill Climbing. 67, 68

ILS Iterated Local Search. 68

LD Levenshtein Distance. 207

LGP Linear based Genetic Programming. 45

MA Metropolis Algorithm. 68

meta-EA Meta-optimization Evolutionary Algorithm. 134

ONM Orthogonal Neighbourhood Mutation. 146

PADO Parallel Algorithm Discovery and Orchestration. 34

PCGP Positional Cartesian Genetic Programming. 47

PDGP Parallel Distributed Genetic Programming. 32

PLGP Parallel linear based Genetic Programming. 45, 46

SA Simulated Annealing. 68, 75

SAGMS Single Active Gene Mutation Strategy. 145

SAW Stepwise adaptation of weights. 45

SD Standard Deviation. 113, 153

SEM Standard Error of the Mean. 153

TS Tabu Search. 68

List of Defintions

List of Algorithms

List of Figures

List of Tables

Bibliography

[1] Peter J. Angeline. An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 21–29, Stanford University, CA, USA, 28–31 July 1996. MIT Press. URL: http://www.natural-selection.com/Library/1996/gp96.zip.

[2] Timothy Atkinson, Detlef Plump, and Susan Stepney. Horizontal gene transfer for recombining graphs. *Genetic Programming and Evolvable Machines*. Special Issue: Highlights of Genetic Programming 2019 Events. doi:doi:10.1007/s10710-020-09378-1.

[3] Timothy Atkinson, Detlef Plump, and Susan Stepney. Evolving graphs by graph programming. In Mauro Castelli, Lukas Sekanina, Mengjie Zhang, Stefano Cagnoni, and Pablo Garcia-Sanchez, editors, *EuroGP 2018: Proceedings of the 21st European Conference on Genetic Programming*, volume 10781 of *LNCS*, pages 35–51, Parma, Italy, 4-6 April 2018. Springer Verlag. URL: http://eprints.whiterose.ac.uk/126500/1/AtkinsonPlumpStepney.EuroGP.18.pdf, doi:doi:10.1007/978-3-319-77553-1_3.

[4] Timothy Atkinson, Detlef Plump, and Susan Stepney. Evolving graphs with horizontal gene transfer. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, page 968–976, New York, NY, USA, 2019. Association for Computing Machinery. URL: https://doi.org/10.1145/3321707.3321788, doi:10.1145/3321707.3321788.

[5] Thomas Bäck, A. E. Eiben, and Marco E. Vink. A superior evolutionary algorithm for 3-sat. In V. William Porto, N. Saravanan, Donald E. Waagen, and A. E. Eiben, editors, *Evolutionary Programming VII, 7th International Conference, EP98, San Diego, CA, USA, March 25-27, 1998, Proceedings*, volume 1447 of *Lecture Notes in Computer Science*, pages 125–136. Springer, 1998. URL: https://doi.org/10.1007/BFb0040766, doi:10.1007/BFb0040766.

[6] Wolfgang Banzhaf. Genetic programming for pedestrians. In *Proceedings of the 5th International Conference on Genetic Algorithms*, page 628, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[7] Wolfgang Banzhaf. Genetic programming for pedestrians. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, page 628, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann. URL: http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/GenProg_forPed.ps.Z.

[8] David F. Barrero, Maria R-Moreno, Bonifacio Castano, and David Camacho. An empirical study on the accuracy of computational effort in genetic programming. In Alice E. Smith, editor, *Proceedings of the 2011 IEEE Congress on Evolutionary Computation*, pages 1169–1176, New Orleans, USA, 5-8 June 2011. IEEE Computational Intelligence Society, IEEE Press. doi:doi:10.1109/CEC.2011.5949748.

[9] Xinye Cai, Stephen L. Smith, and Andy M. Tyrrell. Positional independence and recombination in cartesian genetic programming. In Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 351–360, Budapest, Hungary, 10 - 12 April 2006. Springer. doi:doi:10.1007/11729976_32.

[10] Xinye Cai, Stephen L. Smith, and Andy M. Tyrrell. Positional Independence and Recombination in Cartesian Genetic Programming. In *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 351–360, Budapest, Hungary, 10 - 12 April 2006. Springer. doi:doi:10.1007/11729976_32.

[11] Claude Carlet, Yves Crama, and Peter L. Hammer. Boolean functions for cryptography and error-correcting codes. In *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, pages 257–397. Cambridge University Press, 2010. URL: https://doi.org/10.1017/cbo9780511780448.011, doi:10.1017/cbo9780511780448.011.

[12] Steffen Christensen and Franz Oppacher. An analysis of Koza's computational effort statistic for genetic programming. In James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 182–191, Kinsale, Ireland, 3-5 April 2002. Springer-Verlag. doi:doi:10.1007/3-540-45984-7_18.

[13] Janet Clegg, James Alfred Walker, and Julian Francis Miller. A new crossover technique for cartesian genetic programming. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual*

conference on Genetic and evolutionary computation, volume 2, pages 1580–1587, London, 7-11 July 2007. ACM Press. URL: http://www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1580.pdf, doi:doi:10.1145/1276958.1277276.

[14] Mark Collins. Finding needles in haystacks is harder with neutrality. *Genetic Programming and Evolvable Machines*, 7(2):131–144, August 2006. Special Issue: Best of GECCO 2005. doi:doi:10.1007/s10710-006-9001-y.

[15] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, 24-26 July 1985. URL: http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/icga1985/icga85_cramer.pdf.

[16] Charles Darwin. *On the Origin of Species by Means of Natural Selection*. Murray, London, 1859. or the Preservation of Favored Races in the Struggle for Life.

[17] Lawrence Davis. Adapting operator probabilities in genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms, George Mason University, Fairfax, Virginia, USA, June 1989*, pages 61–69, 1989.

[18] Benjamin Doerr, Daniel Johannsen, and Carola Winzen. Multiplicative drift analysis. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 1449–1456, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1830483.1830748, doi:10.1145/1830483.1830748.

[19] Benjamin Doerr, Daniel Johannsen, and Carola Winzen. Multiplicative drift analysis. *CoRR*, abs/1101.0776, 2011. URL: http://arxiv.org/abs/1101.0776.

[20] Benjamin Doerr, Daniel Johannsen, and Carola Winzen. Multiplicative drift analysis. *Algorithmica*, 64(4):673–697, 2012. URL: https://doi.org/10.1007/s00453-012-9622-x, doi:10.1007/s00453-012-9622-x.

[21] Benjamin Doerr, Andrei Lissovoi, and Pietro Simone Oliveto. Evolving boolean functions with conjunctions and disjunctions via genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 1003–1011, 2019. URL: https://doi.org/10.1145/3321707.3321851, doi:10.1145/3321707.3321851.

[22] Carlton Downey and Mengjie Zhang. Parallel linear genetic programming. In Sara Silva, James A. Foster, Miguel Nicolau, Penousal Machado,

and Mario Giacobini, editors, *Genetic Programming - 14th European Conference, EuroGP 2011, Torino, Italy, April 27-29, 2011. Proceedings*, volume 6621 of *Lecture Notes in Computer Science*, pages 178–189. Springer, 2011. URL: https://doi.org/10.1007/978-3-642-20407-4_16, doi:10.1007/978-3-642-20407-4_16.

[23] Jeroen Eggermont, A. E. Eiben, and Jano I. van Hemert. Adapting the fitness function in GP for data mining. In Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty, editors, *Genetic Programming, Second European Workshop, Göteborg, Sweden, May 26-27, 1999, Proceedings*, volume 1598 of *Lecture Notes in Computer Science*, pages 193–202. Springer, 1999. URL: https://doi.org/10.1007/3-540-48885-5_16, doi:10.1007/3-540-48885-5_16.

[24] Jeroen Eggermont and Jano I. van Hemert. Adaptive genetic programming applied to new and existing simple regression problems. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea Tettamanzi, and William B. Langdon, editors, *Genetic Programming, 4th European Conference, EuroGP 2001, Lake Como, Italy, April 18-20, 2001, Proceedings*, volume 2038 of *Lecture Notes in Computer Science*, pages 23–35. Springer, 2001. URL: https://doi.org/10.1007/3-540-45355-5_3, doi:10.1007/3-540-45355-5_3.

[25] A. E. Eiben, J. K. van der Hauw, and Jano I. van Hemert. Graph coloring with adaptive evolutionary algorithms. *J. Heuristics*, 4(1):25–46, 1998. URL: https://doi.org/10.1023/A:1009638304510, doi:10.1023/A:1009638304510.

[26] A. E. Eiben, Jano I. van Hemert, Elena Marchiori, and Adri G. Steenbeek. Solving binary constraint satisfaction problems using evolutionary algorithms with an adaptive fitness function. In A. E. Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN V, 5th International Conference, Amsterdam, The Netherlands, September 27-30, 1998, Proceedings*, volume 1498 of *Lecture Notes in Computer Science*, pages 201–210. Springer, 1998. URL: https://doi.org/10.1007/BFb0056863, doi:10.1007/BFb0056863.

[27] Lawrence J Fogel. Autonomous automata. *Industrial research*, 4:14–19, 1962.

[28] Lawrence J. Fogel. Toward inductive inference automata. In *Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962*, pages 395–400. North-Holland, 1962.

[29] Lawrence J Fogel, Alvin J Owens, and Michael J Walsh. Artificial intelligence through simulated evolution. 1966.

[30] Richard Forsyth. BEAGLE a Darwinian approach to pattern recognition. *Kybernetes*, 10(3):159–166, 1981. URL: http://www.richardsandesforsyth.net/pubs/beagle81.pdf, doi:doi:10.1108/eb005587.

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

[32] Brian W. Goldman and William F. Punch. Length bias and search limitations in cartesian genetic programming. In Christian Blum, Enrique Alba, Anne Auger, Jaume Bacardit, Josh Bongard, Juergen Branke, Nicolas Bredeche, Dimo Brockhoff, Francisco Chicano, Alan Dorin, Rene Doursat, Aniko Ekart, Tobias Friedrich, Mario Giacobini, Mark Harman, Hitoshi Iba, Christian Igel, Thomas Jansen, Tim Kovacs, Taras Kowaliw, Manuel Lopez-Ibanez, Jose A. Lozano, Gabriel Luque, John McCall, Alberto Moraglio, Alison Motsinger-Reif, Frank Neumann, Gabriela Ochoa, Gustavo Olague, Yew-Soon Ong, Michael E. Palmer, Gisele Lobo Pappa, Konstantinos E. Parsopoulos, Thomas Schmickl, Stephen L. Smith, Christine Solnon, Thomas Stuetzle, El-Ghazali Talbi, Daniel Tauritz, and Leonardo Vanneschi, editors, *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 933–940, Amsterdam, The Netherlands, 6-10 July 2013. ACM. doi:doi:10.1145/2463372.2463482.

[33] Brian W. Goldman and William F. Punch. Reducing wasted evaluations in cartesian genetic programming. In Krzysztof Krawiec, Alberto Moraglio, Ting Hu, A. Sima Uyar, and Bin Hu, editors, *Proceedings of the 16th European Conference on Genetic Programming, EuroGP 2013*, volume 7831 of *LNCS*, pages 61–72, Vienna, Austria, 3-5 April 2013. Springer Verlag. doi:doi:10.1007/978-3-642-37207-0_6.

[34] Brian W. Goldman and William F. Punch. Analysis of cartesian genetic programming's evolutionary mechanisms. *IEEE Transactions on Evolutionary Computation*, 19(3):359–373, 2015. doi:doi:10.1109/TEVC.2014.2324539.

[35] Frédéric Gruau. *On Using Syntactic Constraints with Genetic Programming*, page 377–394. MIT Press, Cambridge, MA, USA, 1996.

[36] Simon Harding, Julian F. Miller, and Wolfgang Banzhaf. Developments in cartesian genetic programming: self-modifying cgp. *Genetic Programming and Evolvable Machines*, 11(3/4):397–439, September 2010. Tenth Anniversary Issue: Progress in Genetic Programming and Evolvable Machines. URL: http://results.ref.ac.uk/Submissions/Output/3354577, doi:doi:10.1007/s10710-010-9114-1.

[37] Robin Harper. Spatial co-evolution: quicker, fitter and less bloated. In Terence Soule and Jason H. Moore, editors, *Genetic and Evolutionary Computation*

Conference, GECCO '12, Philadelphia, PA, USA, July 7-11, 2012, pages 759–766. ACM, 2012. URL: https://doi.org/10.1145/2330163.2330269, doi:10.1145/2330163.2330269.

[38] Trevor Hastie and Robert Tibshirani. Generalized Additive Models. *Statistical Science*, 1(3):297 – 310, 1986. URL: https://doi.org/10.1214/ss/1177013604, doi:10.1214/ss/1177013604.

[39] Trevor Hastie and Robert Tibshirani. *Generalized additive models*, volume 1. Chapman & Hall/CRC Monographs on Statistics and Applied Probability, 1990.

[40] Jun He and Xin Yao. Drift analysis and average time complexity of evolutionary algorithms. *Artif. Intell.*, 127(1):57–85, 2001. URL: https://doi.org/10.1016/S0004-3702(01)00058-3, doi:10.1016/S0004-3702(01)00058-3.

[41] Jun He and Xin Yao. A study of drift analysis for estimating computation time of evolutionary algorithms. *Natural Computing*, 3(1):21–35, 2004. URL: https://doi.org/10.1023/B:NACO.0000023417.31393.c7, doi:10.1023/B:NACO.0000023417.31393.c7.

[42] Joseph Hicklin. Application of the genetic algorithm to automatic program generation. Master's thesis, University of Idaho, 1986.

[43] Nguyen Xuan Hoai, R.I. McKay, and H.A. Abbass. Tree adjoining grammars, language bias, and genetic programming. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming*, pages 335–344, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[44] John H. Holland. Genetic algorithms and the optimal allocation of trials. *SIAM J. Comput.*, 2(2):88–105, 1973. URL: https://doi.org/10.1137/0202009, doi:10.1137/0202009.

[45] John H. Holland. Erratum: Genetic algorithms and the optimal allocation of trials. *SIAM J. Comput.*, 3(4):326, 1974. URL: https://doi.org/10.1137/0203026, doi:10.1137/0203026.

[46] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992. URL: https://doi.org/10.7551/mitpress/1090.001.0001, doi:10.7551/mitpress/1090.001.0001.

[47] Jakub Husa and Roman Kalkreuth. A comparative study on crossover in cartesian genetic programming. In Mauro Castelli, Lukás Sekanina, Mengjie Zhang, Stefano Cagnoni, and Pablo García-Sánchez, editors, *Genetic Programming - 21st European Conference, EuroGP 2018, Parma, Italy, April 4-6, 2018, Proceedings*, volume 10781 of *Lecture Notes in Computer Science*, pages 203–219.

Springer, 2018. URL: https://doi.org/10.1007/978-3-319-77553-1_13, doi:10.1007/978-3-319-77553-1_13.

[48] Fuchuan N I, Yuanxiang L I, and Peng K E. Onmcgp: Orthogonal neighbourhood mutation cartesian genetic programming for evolvable hardware. *Journal of Physics: Conference Series*, 490(1):012194, 2014. URL: http://stacks.iop.org/1742-6596/490/i=1/a=012194.

[49] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation. part i, graph partitioning. *Oper. Res.*, 37(6):865–892, October 1989.

[50] T. Kalganova. Evolutionary approach to design multiple-valued combinational circuits. In *Proceedings. of the 4th International conference on Applications of Computer Systems (ACS'97)*, pages 333–339, Szczecin, Poland, 1997.

[51] Roman Kalkreuth. Two new mutation techniques for cartesian genetic programming. In Juan Julián Merelo Guervós, Jonathan M. Garibaldi, Alejandro Linares-Barranco, Kurosh Madani, and Kevin Warwick, editors, *Proceedings of the 11th International Joint Conference on Computational Intelligence, IJCCI 2019, Vienna, Austria, September 17-19, 2019*, pages 82–92. ScitePress, 2019. URL: https://doi.org/10.5220/0008070100820092, doi:10.5220/0008070100820092.

[52] Roman Kalkreuth. A comprehensive study on subgraph crossover in cartesian genetic programming. In Juan Julián Merelo Guervós, Jonathan M. Garibaldi, Christian Wagner, Thomas Bäck, Kurosh Madani, and Kevin Warwick, editors, *Proceedings of the 12th International Joint Conference on Computational Intelligence, IJCCI 2020, Budapest, Hungary, November 2-4, 2020*, pages 59–70. SCITEPRESS, 2020. URL: https://doi.org/10.5220/0010110700590070, doi:10.5220/0010110700590070.

[53] Roman Kalkreuth. *An Empirical Study on Insertion and Deletion Mutation in Cartesian Genetic Programming*, pages 85–114. Springer International Publishing, Cham, 2021. URL: https://doi.org/10.1007/978-3-030-70594-7_4, doi:10.1007/978-3-030-70594-7_4.

[54] Roman Kalkreuth and Andre Droschinsky. On the time complexity of simple cartesian genetic programming. In Juan Julián Merelo Guervós, Jonathan Garibaldi, Alejandro Linares-Barranco, Kurosh Madani, and Kevin Warwick, editors, *Proceedings of the 11th International Joint Conference on Computational Intelligence, IJCCI 2019, Vienna, Austria, September 17-19, 2019*, pages 172–179. ScitePress, 2019. URL: https://doi.org/10.5220/0008070201720179, doi:10.5220/0008070201720179.

[55] Roman Kalkreuth, Guenter Rudolph, and Jorg Krone. Improving convergence in cartesian genetic programming using adaptive crossover, mutation and selection. In *2015 IEEE Symposium Series on Computational Intelligence*, pages 1415–1422, December 2015. doi:doi:10.1109/SSCI.2015.201.

[56] Roman Kalkreuth, Günter Rudolph, and Andre Droschinsky. A new subgraph crossover for cartesian genetic programming. In James McDermott, Mauro Castelli, Lukás Sekanina, Evert Haasdijk, and Pablo García-Sánchez, editors, *Genetic Programming - 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings*, volume 10196 of *Lecture Notes in Computer Science*, pages 294–310, 2017. URL: https://doi.org/10.1007/978-3-319-55696-3_19, doi:10.1007/978-3-319-55696-3_19.

[57] Roman Kalkreuth, Günter Rudolph, and Jörg Krone. Improving convergence in cartesian genetic programming using adaptive crossover, mutation and selection. In *IEEE Symposium Series on Computational Intelligence, SSCI 2015, Cape Town, South Africa, December 7-10, 2015*, pages 1415–1422. IEEE, 2015. URL: https://doi.org/10.1109/SSCI.2015.201, doi:10.1109/SSCI.2015.201.

[58] Wolfgang Kantschik and Wolfgang Banzhaf. Linear-graph GP - A new GP structure. In James A. Foster, Evelyne Lutton, Julian F. Miller, Conor Ryan, and Andrea Tettamanzi, editors, *Genetic Programming, 5th European Conference, EuroGP 2002, Kinsale, Ireland, April 3-5, 2002, Proceedings*, volume 2278 of *Lecture Notes in Computer Science*, pages 83–92. Springer, 2002. URL: https://doi.org/10.1007/3-540-45984-7_8, doi:10.1007/3-540-45984-7_8.

[59] Paul Kaufmann and Roman Kalkreuth. An empirical study on the parametrization of cartesian genetic programming. In Peter A. N. Bosman, editor, *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, pages 231–232. ACM, 2017. URL: https://doi.org/10.1145/3067695.3075980, doi:10.1145/3067695.3075980.

[60] Paul Kaufmann and Roman Kalkreuth. Parametrizing cartesian genetic programming: An empirical study. In Gabriele Kern-Isberner, Johannes Fürnkranz, and Matthias Thimm, editors, *KI 2017: Advances in Artificial Intelligence - 40th Annual German Conference on AI, Dortmund, Germany, September 25-29, 2017, Proceedings*, volume 10505 of *Lecture Notes in Computer Science*, pages 316–322. Springer, 2017. URL: https://doi.org/10.1007/978-3-319-67190-1_26, doi:10.1007/978-3-319-67190-1_26.

[61] Paul Kaufmann and Marco Platzner. Advanced techniques for the creation and propagation of modules in cartesian genetic programming. In *GECCO*

'08: Proceedings of the 10th annual conference on Genetic and evolutionary computation, pages 1219–1226, Atlanta, GA, USA, 12-16 July 2008. ACM. URL: http://www.cs.bham.ac.uk/~wbl/biblio/gecco2008/docs/p1219.pdf, doi:doi:10.1145/1389095.1389334.

[62] Paul Kaufmann and Marco Platzner. Advanced Techniques for the Creation and Propagation of Modules in Cartesian Genetic Programming. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1219–1226, Atlanta, GA, USA, 12-16 July 2008. ACM. doi:doi:10.1145/1389095.1389334.

[63] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983. URL: https://science.sciencemag.org/content/220/4598/671, arXiv:https://science.sciencemag.org/content/220/4598/671.full.pdf, doi:10.1126/science.220.4598.671.

[64] Michael F. Korns. *Accuracy in Symbolic Regression*, pages 129–151. Springer New York, New York, NY, 2011. URL: https://doi.org/10.1007/978-1-4614-1770-5_8, doi:10.1007/978-1-4614-1770-5_8.

[65] Timo Kötzing, Andrew M. Sutton, Frank Neumann, and Una-May O'Reilly. The max problem revisited: the importance of mutation in genetic programming. In Terence Soule and Jason H. Moore, editors, *Genetic and Evolutionary Computation Conference, GECCO '12, Philadelphia, PA, USA, July 7-11, 2012*, pages 1333–1340. ACM, 2012. URL: https://doi.org/10.1145/2330163.2330348, doi:10.1145/2330163.2330348.

[66] J. Koza. Genetic Programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Dept. of Computer Science, Stanford University, June 1990.

[67] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992. URL: http://mitpress.mit.edu/books/genetic-programming.

[68] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994. URL: http://www.genetic-programming.org/gpbook2toc.html.

[69] Donald H. Kraft, Frederick E. Petry, Bill P. Buckles, and Thyagarajan Sadasivan. The use of genetic programming to build queries for information retrieval. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 468–473, Orlando, Florida, USA, 27-29 June 1994. IEEE Press. doi:doi:10.1109/ICEC.1994.349905.

[70] W. B. Langdon and S. M. Gustafson. Genetic programming and evolvable machines: ten years of reviews. *Genetic Programming and Evolvable Machines*, 11(3/4):321–338, September 2010. Tenth Anniversary Issue: Progress in Genetic Programming and Evolvable Machines. URL: http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/gppubs10.pdf, doi:doi:10.1007/s10710-010-9111-4.

[71] W. B. Langdon and R. Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London, 23-27 June 1997. URL: http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/WBL.bloat_wsc2.ps.gz, doi:doi:10.1007/978-1-4471-0427-8_2.

[72] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002. URL: http://www.cs.ucl.ac.uk/staff/W.Langdon/FOGP/, doi:doi:10.1007/978-3-662-04726-2.

[73] William B. Langdon. Genetically improved software. In Amir H. Gandomi, Amir H. Alavi, and Conor Ryan, editors, *Handbook of Genetic Programming Applications*, chapter 8, pages 181–220. Springer, 2015. URL: http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/langdon_2015_hbgpa.pdf, doi:doi:10.1007/978-3-319-20883-1_8.

[74] William B. Langdon and Adil Qureshi. Genetic programming – computers using "natural selection" to generate programs. In *WC1E 6BT*, 1995.

[75] William B. Langdon, Terry Soule, Riccardo Poli, and James A. Foster. The evolution of size and shape. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA, June 1999. URL: http://www.cs.ucl.ac.uk/staff/W.Langdon/aigp3/ch08.pdf.

[76] Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[77] Andrei Lissovoi and Pietro S. Oliveto. *Computational Complexity Analysis of Genetic Programming*, pages 475–518. Springer International Publishing, Cham, 2020. URL: https://doi.org/10.1007/978-3-030-29414-4_11, doi:10.1007/978-3-030-29414-4_11.

[78] Andrei Lissovoi and Pietro Simone Oliveto. On the time and space complexity of genetic programming for evolving boolean conjunctions. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 1363–1370,

2018. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16145.

[79] M. A. Lones and A. M. Tyrrell. Enzyme genetic programming. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, volume 2, pages 1183–1190 vol. 2, 2001.

[80] Michael A. Lones and Andy M. Tyrell. Enzyme genetic programming. In Martyn Amos, editor, *Cellular Computing*, Series in systems biology, pages 19–42. Oxford University Press, 2004.

[81] Michael A. Lones and Andrew M. Tyrrell. Biomimetic representation with genetic programming enzyme. *Genetic Programming and Evolvable Machines*, 3(2):193–217, 2002. URL: https://doi.org/10.1023/A:1015583926171, doi:10.1023/A:1015583926171.

[82] Michael Adam Lones. *Enzyme genetic programming : modelling biological evolvability in genetic programming*. PhD thesis, University of York, UK, 2003. URL: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.399653.

[83] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43 – 58, 2016. URL: http://www.sciencedirect.com/science/article/pii/S2214716015300270, doi:https://doi.org/10.1016/j.orp.2016.09.002.

[84] Sean Luke. ECJ then and now. In Peter A. N. Bosman, editor, *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, pages 1223–1230. ACM, 2017. URL: https://doi.org/10.1145/3067695.3082467, doi:10.1145/3067695.3082467.

[85] Sean Luke and Liviu Panait. Is the perfect the enemy of the good? In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 820–828, New York, 9-13 July 2002. Morgan Kaufmann Publishers. URL: http://cs.gmu.edu/~sean/papers/ideal.ps.gz.

[86] Sean Luke and Lee Spector. A Comparison of Crossover and Mutation in Genetic Programming. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 240–248, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[87] Sean Luke and Lee Spector. A Revised Comparison of Crossover and Mutation in Genetic Programming. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 208–213, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[88] Andrea Mambrini and Pietro S. Oliveto. On the analysis of simple genetic programming for evolving boolean functions. In Malcolm I. Heywood, James McDermott, Mauro Castelli, Ernesto Costa, and Kevin Sim, editors, *EuroGP 2016: Proceedings of the 19th European Conference on Genetic Programming*, volume 9594 of *LNCS*, pages 99–114, Porto, Portugal, 30 March–1 April 2016. Springer Verlag. doi:doi:10.1007/978-3-319-30668-1_7.

[89] Francisco A. L. Manfrini, Heder S. Bernardino, and Helio J. C. Barbosa. A novel efficient mutation for evolutionary design of combinational logic circuits. In *Parallel Problem Solving from Nature - PPSN XIV - 14th International Conference, Edinburgh, UK, September 17-21, 2016, Proceedings*, pages 665–674, 2016. URL: https://doi.org/10.1007/978-3-319-45823-6_62, doi:10.1007/978-3-319-45823-6_62.

[90] James McDermott, David R. White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, and Una-May O'Reilly. Genetic programming needs better benchmarks. In Terry Soule, Anne Auger, Jason Moore, David Pelta, Christine Solnon, Mike Preuss, Alan Dorin, Yew-Soon Ong, Christian Blum, Dario Landa Silva, Frank Neumann, Tina Yu, Aniko Ekart, Will Browne, Tim Kovacs, Man-Leung Wong, Clara Pizzuti, Jon Rowe, Tobias Friedrich, Giovanni Squillero, Nicolas Bredeche, Stephen L. Smith, Alison Motsinger-Reif, Jose Lozano, Martin Pelikan, Silja Meyer-Nienberg, Christian Igel, Greg Hornby, Rene Doursat, Steve Gustafson, Gustavo Olague, Shin Yoo, John Clark, Gabriela Ochoa, Gisele Pappa, Fernando Lobo, Daniel Tauritz, Jurgen Branke, and Kalyanmoy Deb, editors, *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 791–798, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM. doi:doi:10.1145/2330163.2330273.

[91] Brian McGinley, John Maher, Colm O'Riordan, and Fearghal Morgan. Maintaining healthy population diversity using adaptive crossover, mutation, and selection. *IEEE Trans. Evolutionary Computation*, 15(5):692–714, 2011. URL: https://doi.org/10.1109/TEVC.2010.2046173, doi:10.1109/TEVC.2010.2046173.

[92] Nicholas Freitag McPhee and Justin Darwin Miller. Accurate replication in genetic programming. In Larry J. Eshelman, editor, *Proceedings of the 6th International Conference on Genetic Algorithms, Pittsburgh, PA, USA, July 15-19, 1995*, pages 303–309. Morgan Kaufmann, 1995.

[93] Andreas Meier, Mark Gonter, and Rudolf Kruse. Accelerating convergence in cartesian genetic programming by using a new genetic operator. In Christian Blum, Enrique Alba, Anne Auger, Jaume Bacardit, Josh Bongard, Juergen Branke, Nicolas Bredeche, Dimo Brockhoff, Francisco Chicano, Alan Dorin,

Rene Doursat, Aniko Ekart, Tobias Friedrich, Mario Giacobini, Mark Harman, Hitoshi Iba, Christian Igel, Thomas Jansen, Tim Kovacs, Taras Kowaliw, Manuel Lopez-Ibanez, Jose A. Lozano, Gabriel Luque, John McCall, Alberto Moraglio, Alison Motsinger-Reif, Frank Neumann, Gabriela Ochoa, Gustavo Olague, Yew-Soon Ong, Michael E. Palmer, Gisele Lobo Pappa, Konstantinos E. Parsopoulos, Thomas Schmickl, Stephen L. Smith, Christine Solnon, Thomas Stuetzle, El-Ghazali Talbi, Daniel Tauritz, and Leonardo Vanneschi, editors, *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 981–988, Amsterdam, The Netherlands, 6-10 July 2013. ACM. doi:doi:10.1145/2463372.2463481.

[94] Robert E Mercer and JR Sampson. Adaptive search using a reproductive meta-plan. *Kybernetes*, 1978.

[95] J. F. Miller, P. Thomson, and T. Fogarty. Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study. In *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, pages 105–131. Wiley, 1997.

[96] Julian Miller. What bloat? cartesian genetic programming on Boolean problems. In Erik D. Goodman, editor, *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 295–302, San Francisco, California, USA, 9-11 July 2001. URL: http://www.elec.york.ac.uk/intsys/users/jfm7/gecco2001Late.pdf.

[97] Julian F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1135–1142, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann. URL: http://citeseer.ist.psu.edu/153431.html.

[98] Julian F. Miller, editor. *Cartesian Genetic Programming*. Natural Computing Series. Springer, 2011. URL: https://doi.org/10.1007/978-3-642-17310-3, doi:10.1007/978-3-642-17310-3.

[99] Julian F. Miller and Stephen L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, April 2006. doi:doi:10.1109/TEVC.2006.871253.

[100] Julian F. Miller and Peter Thomson. Cartesian Genetic Programming. In *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 April 2000. Springer-Verlag. doi:doi:10.1007/978-3-540-46239-2_9.

[101] Julian F. Miller and Peter Thomson. Cartesian genetic programming. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 April 2000. Springer-Verlag. URL: http://www.elec.york.ac.uk/intsys/users/jfm7/cgp-eurogp2000.pdf, doi:doi:10.1007/978-3-540-46239-2_9.

[102] Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson. Geometric semantic genetic programming. In Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone, editors, *Parallel Problem Solving from Nature, PPSN XII (part 1)*, volume 7491 of *Lecture Notes in Computer Science*, pages 21–31, Taormina, Italy, September 1-5 2012. Springer. doi:doi:10.1007/978-3-642-32937-1_3.

[103] Alberto Moraglio and Andrea Mambrini. Runtime analysis of mutation-based geometric semantic genetic programming on boolean functions. In Christian Blum, Enrique Alba, Anne Auger, Jaume Bacardit, Josh Bongard, Juergen Branke, Nicolas Bredeche, Dimo Brockhoff, Francisco Chicano, Alan Dorin, Rene Doursat, Aniko Ekart, Tobias Friedrich, Mario Giacobini, Mark Harman, Hitoshi Iba, Christian Igel, Thomas Jansen, Tim Kovacs, Taras Kowaliw, Manuel Lopez-Ibanez, Jose A. Lozano, Gabriel Luque, John McCall, Alberto Moraglio, Alison Motsinger-Reif, Frank Neumann, Gabriela Ochoa, Gustavo Olague, Yew-Soon Ong, Michael E. Palmer, Gisele Lobo Pappa, Konstantinos E. Parsopoulos, Thomas Schmickl, Stephen L. Smith, Christine Solnon, Thomas Stuetzle, El-Ghazali Talbi, Daniel Tauritz, and Leonardo Vanneschi, editors, *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 989–996, Amsterdam, The Netherlands, 6-10 July 2013. ACM. doi:doi:10.1145/2463372.2463492.

[104] Alberto Moraglio, Andrea Mambrini, and Luca Manzoni. Runtime analysis of mutation-based geometric semantic genetic programming on boolean functions. In Frank Neumann and Kenneth De Jong, editors, *Foundations of Genetic Algorithms*, pages 119–132, Adelaide, Australia, 16-20 January 2013. ACM. URL: http://www.cs.bham.ac.uk/~axm322/pdf/gsgp_foga13.pdf, doi:doi:10.1145/2460239.2460251.

[105] Heinz Mühlenbein and Dirk Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm i. continuous parameter optimization. *Evol. Comput.*, 1(1):25–49, March 1993. URL: https://doi.org/10.1162/evco.1993.1.1.25, doi:10.1162/evco.1993.1.1.25.

[106] Barricelli NA. Esempi numerici di processi di evoluzion. *Methodos*, 6(21-22):45–68, 1954.

[107] Frank Neumann, Una-May O'Reilly, and Markus Wagner. *Computational Complexity Analysis of Genetic Programming - Initial Results and Fu-*

ture Directions, pages 113–128. Springer New York, New York, NY, 2011. URL: https://doi.org/10.1007/978-1-4614-1770-5_7, doi:10.1007/978-1-4614-1770-5_7.

[108] Fuchuan Ni, Yuanxiang Li, Xiaoyan Yang, and Jinhai Xiang. An orthogonal cartesian genetic programming algorithm for evolvable hardware. In *2014 International Conference on Identification, Information and Knowledge in the Internet of Things (IIKI)*, pages 220–224, October 2014. doi:doi:10.1109/IIKI.2014.52.

[109] Jens Niehaus and Wolfgang Banzhaf. More on computational effort statistics for genetic programming. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 164–172, Essex, 14-16 April 2003. Springer-Verlag. URL: http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2610&spage=164, doi:doi:10.1007/3-540-36599-0_15.

[110] Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11(3/4):339–363, September 2010. Tenth Anniversary Issue: Progress in Genetic Programming and Evolvable Machines. doi:doi:10.1007/s10710-010-9113-2.

[111] S. Openshaw and I. Turton. Building new spatial interaction models using genetic programming. In *Evolutionary Computing, Lecture Notes in Computer Science*, pages 11–13. Springer-Verlag, 1994.

[112] Ludo Pagie and Paulien Hogeweg. Evolutionary consequences of coevolving targets. *Evol. Comput.*, 5(4):401–418, 1997. URL: https://doi.org/10.1162/evco.1997.5.4.401, doi:10.1162/evco.1997.5.4.401.

[113] Norman Paterson and Michael Livesey. Performance comparison in genetic programming. In Darrell Whitley, editor, *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, pages 253–260, Las Vegas, Nevada, USA, 8 July 2000. URL: http://www.dcs.st-and.ac.uk/~norman/Pubs/PerfCompInGP.ps.gz.

[114] Tim Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153, Orlando, Florida, USA, 27-29 June 1994. IEEE Press. URL: http://citeseer.ist.psu.edu/432690.html, doi:doi:10.1109/ICEC.1994.350025.

[115] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. Genetic improvement of software: a comprehensive survey. *IEEE Transactions*

on Evolutionary Computation, 22(3):415–432, June 2018. URL: http://www.cs.ucl.ac.uk/staff/J.Petke/papers/Petke_2017_TEVC.pdf, doi:doi:10.1109/TEVC.2017.2693219.

[116] Stjepan Picek, Domagoj Jakobovic, Julian F. Miller, Lejla Batina, and Marko Cupic. Cryptographic boolean functions: One output, many design criteria. *Appl. Soft Comput.*, 40:635–653, 2016. URL: https://doi.org/10.1016/j.asoc.2015.10.066, doi:10.1016/j.asoc.2015.10.066.

[117] Martin Pincus. Letter to the editor—a monte carlo method for the approximate solution of certain types of constrained optimization problems. *Operations research*, 18(6):1225–1228, 1970.

[118] Detlef Plump. The graph programming language GP. In Symeon Bozapalidis and George Rahonis, editors, *Algebraic Informatics, Third International Conference, CAI 2009, Thessaloniki, Greece, May 19-22, 2009, Proceedings*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer, 2009. URL: https://doi.org/10.1007/978-3-642-03564-7_6, doi:10.1007/978-3-642-03564-7_6.

[119] Detlef Plump. The design of GP 2. In Santiago Escobar, editor, *Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2011, Novi Sad, Serbia, 29 May 2011*, volume 82 of *EPTCS*, pages 1–16, 2011. URL: https://doi.org/10.4204/EPTCS.82.1, doi:10.4204/EPTCS.82.1.

[120] R. Poli. Some steps towards a form of parallel distributed genetic programming. In *The 1st Online Workshop on Soft Computing (WSC1)*, http://www.bioele.nuee.nagoya-u.ac.jp/wsc1/, 19–30 August 1996. Nagoya University, Japan. URL: ftp://ftp.cs.bham.ac.uk/pub/authors/R.Poli/wsc96.ps.gz.

[121] Riccardo Poli. Parallel distributed genetic programming. Technical Report CSRP-96-15, School of Computer Science, University of Birmingham, B15 2TT, UK, September 1996. URL: ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1996/CSRP-96-15.ps.gz.

[122] Riccardo Poli. Evolution of graph-like programs with parallel distributed genetic programming. In Thomas Back, editor, *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 346–353, Michigan State University, East Lansing, MI, USA, 19-23 July 1997. Morgan Kaufmann. URL: http://cswww.essex.ac.uk/staff/rpoli/papers/Poli-ICGA1997-PDGP.pdf.

[123] Riccardo Poli. Parallel distributed genetic programming applied to the evolution of natural language recognisers. *Evolutionary Machine Learning and*

Classifier Systems, pages 163–177, 1997. URL: https://doi.org/10.1007/BFb0027173, doi:10.1007/BFb0027173.

[124] Riccardo Poli. *New Ideas in Optimization*, chapter Parallel Distributed Genetic Programming, pages 403–432. McGraw-Hill Ltd., UK, Maidenhead, UK, England, 1999. URL: http://dl.acm.org/citation.cfm?id=329055.329092.

[125] Riccardo Poli. General schema theory for genetic programming with subtree-swapping crossover. Technical Report CSRP-00-16, University of Birmingham, School of Computer Science, November 2000. URL: ftp://ftp.cs.bham.ac.uk/pub/tech-reports/2000/CSRP-00-16.ps.gz.

[126] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008. (With contributions by J. R. Koza). URL: http://www.gp-field-guide.org.uk.

[127] Riccardo Poli and Nicholas Freitag McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part I. *Evolutionary Computation*, 11(1):53–66, March 2003. URL: http://cswww.essex.ac.uk/staff/rpoli/papers/ecj2003partI.pdf, doi:doi:10.1162/106365603321829005.

[128] Riccardo Poli and Nicholas Freitag McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part II. *Evolutionary Computation*, 11(2):169–206, June 2003. URL: http://cswww.essex.ac.uk/staff/rpoli/papers/ecj2003partII.pdf, doi:doi:10.1162/106365603766646825.

[129] Riccardo Poli, Nicholas Freitag McPhee, and Jonathan E. Rowe. Exact schema theory and markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover. *Genetic Programming and Evolvable Machines*, 5(1):31–70, 2004. URL: https://doi.org/10.1023/B:GENP.0000017010.41337.a7, doi:10.1023/B:GENP.0000017010.41337.a7.

[130] I. Rechenberg. Cybernetic solution path of an experimental problem. In *Royal Aircraft Establishment Translation No. 1122, B. F. Toms, Trans.* Ministry of Aviation, Royal Aircraft Establishment, Farnborough Hants, August 1965.

[131] Ingo Rechenberg. *Evolutionsstrategie Optimierung technischer Systeme nach Prinzipien der biologishen Evolution*. Frommann Holzboog Verlag, Stuttgart, 1973.

[132] Patricia Ryser-Welch, Julian F. Miller, Jerry Swan, and Martin A. Trefzer. Iterative cartesian genetic programming: Creating general algorithms for solving travelling salesman problems. In Malcolm I. Heywood, James McDermott,

Mauro Castelli, Ernesto Costa, and Kevin Sim, editors, *EuroGP 2016: Proceedings of the 19th European Conference on Genetic Programming*, volume 9594 of *LNCS*, pages 294–310, Porto, Portugal, 30 March–1 April 2016. Springer Verlag. doi:doi:10.1007/978-3-319-30668-1_19.

[133] Palash Sarkar and Subhamoy Maitra. Nonlinearity bounds and constructions of resilient boolean functions. In *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, pages 515–532, 2000. URL: https://doi.org/10.1007/3-540-44598-6_32, doi:10.1007/3-540-44598-6_32.

[134] J. David Schaffer and Amy Morishima. An adaptive crossover distribution mechanism for genetic algorithms. In *Proceedings of the 2nd International Conference on Genetic Algorithms, Cambridge, MA, USA, July 1987*, pages 36–40, 1987.

[135] Hans-Paul Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie: mit einer vergleichenden Einführung in die Hill-Climbing-und Zufallsstrategie*, volume 1. Springer, 1977.

[136] Eric O. Scott and Sean Luke. ECJ at 20: toward a general metaheuristics toolkit. In Manuel López-Ibáñez, Anne Auger, and Thomas Stützle, editors, *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 1391–1398. ACM, 2019. URL: https://doi.org/10.1145/3319619.3326865, doi:10.1145/3319619.3326865.

[137] Lukás Sekanina. Image filter design with evolvable hardware. In *Applications of Evolutionary Computing, EvoWorkshops 2002: EvoCOP, EvoIASP, EvoSTIM/EvoPLAN, Kinsale, Ireland, April 3-4, 2002, Proceedings*, pages 255–266, 2002. URL: https://doi.org/10.1007/3-540-46004-7_26, doi:10.1007/3-540-46004-7_26.

[138] Karel Slaný and Lukás Sekanina. Fitness landscape analysis and image filter evolution using functional-level cgp. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 311–320, Valencia, Spain, 11-13 April 2007. Springer. doi:doi:10.1007/978-3-540-71605-1_29.

[139] Stephen L. Smith, Stefan Leggett, and Andrew M. Tyrrell. An implicit context representation for evolving image processing filters. In Franz Rothlauf, Jürgen Branke, Stefano Cagnoni, David Wolfe Corne, Rolf Drechsler, Yaochu Jin, Penousal Machado, Elena Marchiori, Juan Romero, George D. Smith, and Giovanni Squillero, editors, *Applications of Evolutionary Computing*, pages 407–416, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[140] Terence Soule and James A. Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *1998 IEEE International Conference on Evolutionary Computation*, pages 781–786, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press. URL: http://citeseer.ist.psu.edu/313655.html, doi:doi:10.1109/ICEC.1998.700151.

[141] Lee Spector. Autoconstructive evolution: Push, pushgp, and pushpop. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann. URL: http://hampshire.edu/lspector/pubs/ace.pdf.

[142] Lee Spector and Alan J. Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, 2002. URL: https://doi.org/10.1023/A:1014538503543, doi:10.1023/A:1014538503543.

[143] M. Srinivas and Lalit M. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Trans. Systems, Man, and Cybernetics*, 24(4):656–667, 1994. URL: https://doi.org/10.1109/21.286385, doi:10.1109/21.286385.

[144] Astro Teller and Manuela Veloso. Pado: A new learning architecture for object recognition. In Katsushi Ikeuchi and Manuela Veloso, editors, *Symbolic Visual Learning*, pages 81–116. Oxford University Press, 1996. URL: http://www.cs.cmu.edu/afs/cs/usr/astro/public/papers/PADO.ps.

[145] Andrew Turner and Julian Miller. Cartesian genetic programming: Why no bloat? In Miguel Nicolau, Krzysztof Krawiec, Malcolm I. Heywood, Mauro Castelli, Pablo Garcia-Sanchez, Juan J. Merelo, Victor M. Rivas Santos, and Kevin Sim, editors, *17th European Conference on Genetic Programming*, volume 8599 of *LNCS*, pages 222–233, Granada, Spain, 23-25 April 2014. Springer. doi:doi:10.1007/978-3-662-44303-3_19.

[146] Andrew Turner and Julian Miller. Recurrent cartesian genetic programming. In Thomas Bartz-Beielstein, Juergen Branke, Bogdan Filipic, and Jim Smith, editors, *13th International Conference on Parallel Problem Solving from Nature*, volume 8672 of *Lecture Notes in Computer Science*, pages 476–486, Ljubljana, Slovenia, 13-17 September 2014. Springer. doi:doi:10.1007/978-3-319-10762-2_47.

[147] Andrew James Turner. Improving crossover techniques in a genetic program. Master's thesis, Department of Electronics, University of York, 2012.

[148] Andrew James Turner and Julian Francis Miller. Cartesian genetic programming encoded artificial neural networks: a comparison using three benchmarks. In Christian Blum, Enrique Alba, Anne Auger, Jaume Bacardit, Josh Bongard, Juergen Branke, Nicolas Bredeche, Dimo Brockhoff, Francisco Chicano, Alan Dorin, Rene Doursat, Aniko Ekart, Tobias Friedrich, Mario Giacobini, Mark Harman, Hitoshi Iba, Christian Igel, Thomas Jansen, Tim Kovacs, Taras Kowaliw, Manuel Lopez-Ibanez, Jose A. Lozano, Gabriel Luque, John McCall, Alberto Moraglio, Alison Motsinger-Reif, Frank Neumann, Gabriela Ochoa, Gustavo Olague, Yew-Soon Ong, Michael E. Palmer, Gisele Lobo Pappa, Konstantinos E. Parsopoulos, Thomas Schmickl, Stephen L. Smith, Christine Solnon, Thomas Stuetzle, El-Ghazali Talbi, Daniel Tauritz, and Leonardo Vanneschi, editors, *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 1005–1012, Amsterdam, The Netherlands, 6-10 July 2013. ACM. doi:doi:10.1145/2463372.2463484.

[149] Andrew James Turner and Julian Francis Miller. The importance of topology evolution in neuroevolution: A case study using cartesian genetic programming of artificial neural networks. In Max Bramer and Miltos Petridis, editors, *Research and Development in Intelligent Systems XXX*, pages 213–226, Cham, 2013. Springer International Publishing.

[150] Andrew James Turner and Julian Francis Miller. Neutral genetic drift: an investigation using cartesian genetic programming. *Genetic Programming and Evolvable Machines*, 16(4):531–558, December 2015. doi:doi:10.1007/s10710-015-9244-6.

[151] Andrew James Turner and Julian Francis Miller. Recurrent cartesian genetic programming of artificial neural networks. *Genetic Programming and Evolvable Machines*, 18(2):185–212, June 2017. doi:doi:10.1007/s10710-016-9276-6.

[152] Vesselin K. Vassilev and Julian F. Miller. The advantages of landscape neutrality in digital circuit evolution. In *Proceedings of the Third International Conference on Evolvable Systems*, pages 252–263. Springer-Verlag, 2000.

[153] V. Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *J. Optim. Theory Appl.*, 45(1):41–51, January 1985. URL: https://doi.org/10.1007/BF00940812, doi:10.1007/BF00940812.

[154] Ekaterina Vladislavleva, Guido Smits, and Dick den Hertog. 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. URL: https://doi.org/10.1109/TEVC.2008.926486, doi:10.1109/TEVC.2008.926486.

[155] James Alfred Walker and Julian Francis Miller. *Evolution and Acquisition of Modules in Cartesian Genetic Programming*, pages 187–197. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. URL: https://doi.org/10.1007/978-3-540-24650-3_17, doi:10.1007/978-3-540-24650-3_17.

[156] James Alfred Walker and Julian Francis Miller. Evolution and acquisition of modules in cartesian genetic programming. In *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 187–197, Coimbra, Portugal, 5-7 April 2004. Springer-Verlag. URL: http://www.elec.york.ac.uk/intsys/users/jfm7/eurogp2004.pdf, doi:doi:10.1007/978-3-540-24650-3_17.

[157] James Alfred Walker and Julian Francis Miller. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417, August 2008. URL: http://results.ref.ac.uk/Submissions/Output/3354578, doi:doi:10.1109/TEVC.2007.903549.

[158] James Alfred Walker, Julian Francis Miller, and Rachel Cavill. A multi-chromosome approach to standard and embedded cartesian genetic programming. In Maarten Keijzer, Mike Cattolico, Dirk Arnold, Vladan Babovic, Christian Blum, Peter Bosman, Martin V. Butz, Carlos Coello Coello, Dipankar Dasgupta, Sevan G. Ficici, James Foster, Arturo Hernandez-Aguirre, Greg Hornby, Hod Lipson, Phil McMinn, Jason Moore, Guenther Raidl, Franz Rothlauf, Conor Ryan, and Dirk Thierens, editors, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 1, pages 903–910, Seattle, Washington, USA, 8-12 July 2006. ACM Press. URL: http://www.cs.bham.ac.uk/~wbl/biblio/gecco2006/docs/p903.pdf, doi:doi:10.1145/1143997.1144153.

[159] P. A. Whigham. Search bias, language bias and genetic programming. In *Proceedings of the 1st Annual Conference on Genetic Programming*, page 230–237, Cambridge, MA, USA, 1996. MIT Press.

[160] David R. White, James McDermott, Mauro Castelli, Luca Manzoni, Brian W. Goldman, Gabriel Kronberger, Wojciech Jaskowski, Una-May O'Reilly, and Sean Luke. Better GP Benchmarks: Community Survey Results and Proposals. *Genetic Programming and Evolvable Machines*, 14(1):3–29, March 2013. doi:doi:10.1007/s10710-012-9177-2.

[161] David R. White and Simon Poulding. A Rigorous Evaluation of Crossover and Mutation in Genetic Programming. In *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009*, volume 5481 of *LNCS*, pages 220–231, Tuebingen, April 15-17 2009. Springer. doi:doi:10.1007/978-3-642-01181-8_19.

[162] Dennis G. Wilson, Julian F. Miller, Sylvain Cussat-Blanc, and Hervé Luga. Positional cartesian genetic programming. *CoRR*, abs/1810.04119, 2018. URL: http://arxiv.org/abs/1810.04119, arXiv:1810.04119.

[163] Man Leung Wong and Kwong Sak Leung. Inducing logic programs with genetic algorithms: The genetic logic programming system. *IEEE Expert: Intelligent Systems and Their Applications*, 10(5):68–76, October 1995. URL: http://dx.doi.org/10.1109/64.464935, doi:10.1109/64.464935.

[164] Man Leung Wong and Kwong Sak Leung. Evolving recursive functions for the even-parity problem using genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA, 1996. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6277537.

[165] John R. Woodward. Complexity and cartesian genetic programming. In Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 260–269, Budapest, Hungary, 10 - 12 April 2006. Springer. doi:doi:10.1007/11729976_23.

[166] Tina Yu and Julian Miller. Neutrality and the evolvability of Boolean function landscape. In *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 204–217, Lake Como, Italy, 18-20 April 2001. Springer-Verlag. URL: http://www.cs.mun.ca/~tinayu/index_files/addr/public_html/neutrality.pdf.

[167] Tina Yu and Julian F. Miller. Finding needles in haystacks is not hard with neutrality. In *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 13–25, Kinsale, Ireland, 3-5 April 2002. Springer-Verlag. URL: http://www.cs.mun.ca/~tinayu/index_files/addr/public_html/EuroGP2002.pdf, doi:doi:10.1007/3-540-45984-7_2.