Program Synthesis with Grammars and Semantics in Genetic Programming

Stefan Forstenlechner

UCD student number: 14204817

The thesis is submitted to University College Dublin
in fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

School of Business

Head of School:

Prof. Anthony Brabazon

Research Supervisors:

Prof. Michael O'Neill

Dr. Miguel Nicolau

External Examiner:

Dr. John R. Woodward

January 2019

Contents

Abstract

Program synthesis is an important field that has many use cases like bug fixing, automating repetitive tasks and discovering new algorithms. One way to approach program synthesis tasks is to specify a grammar that defines all possible programs that can be created and using a search algorithm like genetic programming to create a program. Although using grammars has the advantage that created programs are syntactically correct, the grammar has to be defined for each problem tackled.

The focus of this thesis is to introduce a grammar design approach that provides the ability to tackle arbitrary program synthesis problems from input/output examples. The grammars will not be required to be tailored to a specific problem, and in contrast to many existing approaches, the code of the produced programs will be in a programming language used by practitioners. The grammar design approach is studied on a range of program synthesis problems throughout the thesis and shows results that are competitive to state of the art systems.

As the search for programs with genetic programming is often done on the syntactic representation without considering the behaviour or semantics of a program, the introduction of semantic operators for program synthesis will be investigated. While in other problem domains, semantic operators have improved search performance, no such operators are available for the program synthesis domain. A definition of semantics in program synthesis will be provided, and multiple semantic measures and operators will be studied on the basis of this definition. The results show that novel semantic crossover and mutation operators for genetic programming can outperform traditional operators that do not consider semantic information.

Statement of Original Authorship

I hereby certify that the submitted work is my own work, was completed while registered as a candidate for the degree stated on the Title Page, and I have not obtained a degree elsewhere on the basis of the research presented in this submitted work

Acknowledgements

First I would like to thank all the members of the NCRA group that have supported me in so many ways. In particular, thank you to my supervisors Michael O'Neill and Miguel Nicolau for giving me the opportunity to complete a PhD and for all their guidance during this time. I also want to say thank you to David Fagan with whom I have discussed most of my ideas and who has given me great feedback. Thank you, David Lynch, Roisin Loughran and James McDermott and everybody who made these past four years in Ireland such an enjoyable time. It was a pleasure to meet and work with you, and I hope to see you in the future.

I would also like to thank my family, who have always been there for me. In particular thank you mum: "Vielen Dank für alles was du für mich getan hast. Ich hätte es nie so weit geschafft, wenn du mich nicht immer bei allem, was ich getan habe, unterstützt hättest. Es war nicht immer leicht, aber du hast immer gewusst das ich es schaffen werde, auch wenn ich gezweifelt habe."

Thank you to all my friends who probably do not even realise how much they have helped and supported me over the years. I hope I can do the same for them.

A special thank you to my fiancée Carina who moved with me to Ireland despite the weather, who kept me going when I had enough but most importantly distracted me from all the stress just by being her joyful self. I could not have done it without you.

Finally, I would like to thank Science Foundation Ireland who supported this research under grant 13/IA/1850.

To my mother Rosemarie Forstenlechner

In memory of my father Kurt Forstenlechner (1960–2009)

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AST	Abstract Syntax Tree
BNF	Backus–Naur Form
CFG	Context-Free Grammar
CFG-GP	Context-Free Grammar Genetic Programming
CRISP-DM	CRoss-Industry Standard Process for Data Mining
CSG	Context-Sensitive Grammar
EA	Evolutionary Algorithm
EC	Evolutionary Computation
EP	Evolutionary Programming
ES	Evolutionary Strategy
ESCPS	Effective Semantic Crossover for Program Synthesis
ESMPS	Effective Semantic Mutation for Program Synthesis
G3P	Grammar-Guided Genetic Programming
GA	Genetic Algorithm
GE	Grammatical Evolution
GI	Genetic Improvement
GP	Genetic Programming
GPPS	Genetic Programming Problem Solver
GSGP	Geometric Semantic Genetic Programming
MSSC	Most Semantically Similar Crossover
PTC2	Probabilistic Tree-Creation 2
ROBDD	Reduced Ordered Binary Decision Diagram
SAM	Semantic Aware Mutation
SASE	Self-Adaptive Successful Execution
SBSE	Search Based Software Engineering
SCPS	Semantic Crossover for Program Synthesis
SCS	Semantic-Clustering Selection

SDC	Semantically Driven Crossover
SDM	Semantically Driven Mutation
SFGP	Strongly Formed Genetic Programming
SiS	Semantics in Selection
SMT	Satisfiability Modulo Theory
SPOT	Sequential Parameter Optimization Toolbox
SSC	Semantic Similarity-based Crossover
SSM	Semantic Similarity-based Mutation
STGP	Strongly Typed Genetic Programming
STS	Semantic Tournament Selection

Publications Arising

1. *S. Forstenlechner*, M. Nicolau, D. Fagan, and M. O'Neill, "Introducing Semantic-Clustering Selection in Grammatical Evolution," in *GECCO 2015 Semantic Methods in Genetic Programming (SMGP'15) Workshop* (C. Johnson, K. Krawiec, A. Moraglio, and M. O'Neill, eds.), (Madrid, Spain), pp. 1277–1284, ACM, 11-15 July 2015.

2. *S. Forstenlechner*, M. Nicolau, D. Fagan, and M. O'Neill, "Grammar Design for Derivation Tree Based Genetic Programming Systems," in *EuroGP 2016: Proceedings of the 19th European Conference on Genetic Programming* (M. I. Heywood, J. McDermott, M. Castelli, and E. Costa, eds.), vol. 9594 of LNCS, (Porto, Portugal), pp. 192–207, Springer Verlag, 30 Mar.–1 Apr. 2016.

3. *S. Forstenlechner*, D. Fagan, M. Nicolau, and M. O'Neill, "A Grammar Design Pattern for Arbitrary Program Synthesis Problems in Genetic Programming," in *EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming* (M. Castelli, J. McDermott, and L. Sekanina, eds.), vol. 10196 of LNCS, (Amsterdam, Netherlands), pp. 262–277, Springer Verlag, 19-21 Apr. 2017.

4. M. Fenton, J. McDermott, D. Fagan, *S. Forstenlechner*, E. Hemberg, and M. O'Neill, "PonyGE2: Grammatical Evolution in Python," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, (Berlin, Germany), pp. 1194–1201, ACM, 2017.

5. *S. Forstenlechner*, D. Fagan, M. Nicolau, and M. O'Neill, "Semantics-Based Crossover for Program Synthesis in Genetic Programming," in *Artifcial Evolution* (E. Lutton, P. Legrand, P. Parrend, N. Monmarché, and M. Schoenauer, eds.), (Paris, France), pp. 58-71, Springer Verlag, 25-27 Oct. 2017.

6. *S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill*, "Towards Understanding and Refining the General Program Synthesis Benchmark Suite with Genetic Programming," in *CEC 2018: IEEE Congress on Evolutionary Computation*, (Rio de Janeiro, Brasil), IEEE, 8–13 July 2018.

7. *S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill*, "Towards Effective Semantic Operators for Program Synthesis in Genetic Programming," in *GECCO '18: Genetic and Evolutionary Computation Conference*, (Kyoto, Japan), pp. 1119–1126, ACM, 15–19 July 2018.

8. *S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill*, "Extending Program Synthesis Grammars for Grammar-Guided Genetic Programming," in *Parallel Problem Solving from Nature – PPSN XV*, (Coimbra, Portugal), pp. 197–208, Springer Verlag, 8–12 Sep. 2018.

Part I

Introduction and Literature Review

Chapter 1

Introduction

Automatically discovering executable programs (Program Synthesis) has many real-world applications for a wide range of users. Program synthesis can help experienced programmers discover new algorithms [2] or new ways to approach a problem, as well as support them in everyday work by synthesising code for mundane tasks. Even automatic bug fixing is within the applications of program synthesise [3, 4, 5]. Program synthesis can also help users with little or no programming experience carry out repetitive tasks [6].

For example, business analysts face many challenges when it comes to managing, processing and modeling data as well as using data to make predictions. Many tasks after being defined by a business analyst could be automated to free up time for more important work that computers cannot yet handle automatically. A step in the CRoss-Industry Standard Process for Data Mining (CRISP-DM) [7] that can take up much time is data preparation. Data preparation is the phase of converting initial raw data into a final dataset. A solution for users with little or no programming experience could be to prepare a few data points themselves and show a computer the examples. A machine learning technique could synthesize a program that executes the data transformations itself by learning from the samples provided by the user. Even machine learning itself becomes automated (AutoML), to create predictive models automatically, without the need of an expert in the area [8].

As program synthesis is an interesting but also difficult task, research on program synthesis is conducted in many research areas, like Inductive Programming [9, 10], Version Space Learning [11, 6] or Evolutionary Algorithms [12, 13].

Genetic Programming (GP), a form of evolutionary algorithms, is widely used to tackle a variety of problems due to being a very general concept. GP searches for a correct program by using a set of possible solutions and incrementally improving them. While one of GP's goals always was and still is to evolve programs, in most research the programs are restricted to particular forms to keep the search space small, like in symbolic regression. These restrictions often exclude programs in the most general sense, e.g. control flow like conditionals and iterations as well as memory to read and write data. GP has been applied to a range of program synthesis tasks [12, 13, 14], but it and evolution as a paradigm have also been criticised as not being a good paradigm for program induction and that "GP in its current form is fundamentally flawed"[15]. The criticism is mainly focused on the way GP operates, as it makes random changes to the syntax "hoping that improvements in the semantics will result".

Semantics can be defined as "the behavior of a program, once it is executed on a set of data" [16]. Semantic information has been applied in GP in problem domains such as regression and boolean. Different approaches to directly or indirectly influence the behaviour of solutions have been researched [16], but semantics has barely been used in the program synthesis domain. Using semantics in program synthesis is more complex than in the regression and boolean domain. While the semantics in the regression and boolean domain is merely a vector of either numeric or boolean values, in program synthesis this is not the case. Program synthesis uses a range of different data types as well as data structures, which already makes the definition of semantics more complicated let alone the usage. As semantics is problem specific, it has to be defined in each problem domain, and operators used for searching have to be adapted.

1.1 Aim of Thesis

This thesis propose a grammar design approach that is capable of generating general purpose programs from input/output examples with the help of a Grammar-Guided Genetic Programming (G3P) system. The purpose of the grammar is to produce code in a programming language used by practitioners, instead of some self-defined language or pseudocode, so that the evolved code can directly be used in a real-world system. The benefit of grammars is that they can be used with any G3P system, therefore be exchanged between researchers, and tested with

different types of G3P systems. The idea is to have reusable grammars that can tackle arbitrary program synthesis problems without the need to be tailored to a specific problem. An analysis of the capabilities of the grammar design approach compared to a state of the art method is conducted, and the goal is to achieve better or at least competitive results. The proposed approach will be analysed on a set of benchmark problems and limitations will be pointed out as well as suggestions for improvements.

While previous G3P systems focused on bias [17] by, e.g. adding expert knowledge in grammars or having a biologically inspired approach to evolve solutions in arbitrary languages [18], the grammar design approach aims to tackle arbitrary problems in arbitrary languages. The grammar design approach describes how to produce grammars to tackle arbitrary program synthesis problems. At the same time, the approach should be applicable to a number of programming languages.

A proposal for semantics in program synthesis will be made to introduce novel operators that make changes based on the behaviour of a program rather than random syntactic ones to improve solutions. The semantic operators will be compared to ones operating on syntax to prove their advantages.

1.2 Research Questions

The research questions focus on two topics. The first topic is the use of grammars in GP to tackle program synthesis and how grammars that produce different derivation tree structures influence performance. The second topic addresses the lack of utilization of semantics in program synthesis to date. Therefore, a goal of this thesis will be focusing on the improvement of GP in solving program synthesis problems by integrating semantic information in the search process. The research questions going to be addressed are listed below.

Question 1: *How can Grammar-Guided Genetic Programming (G3P) be utilized to tackle program synthesis?*

Program synthesis is a problem domain to which G3P seems especially well suited for the following reasons. 1) Programming languages are already defined in grammars as grammars are used to check syntactical correctness. 2) The same or similar problems have to be solved in many programming

languages. As many programming language have similar syntactical structure, it should be easy to adapt grammars from one language to another. 3) Grammars can be exchanged between researchers and used in all kinds of GP systems without reimplementing code. Therefore, experiments can be reproduced in an easy way.

The major drawback of previously used grammars to tackle program synthesis so far is that they have been written for a specific problem instance [13, 19]. Therefore they cannot be reused for another problem instance without adaptation, as they only use certain data types or data structures and just use a small subset of the available functionality of a programming language. These grammars limit the search space but are not applicable to a wide variety of problems. In Chapter 4, a grammar design approach for the creation of grammars for programming languages is presented so that arbitrary program synthesis problems can be tackled, that the grammars can be extended to use any libraries already existing in a language, and that invalid individuals, due to runtime errors etc., are kept to a minimum to avoid wasting resources. The grammars can be used in any G3P system with minimal implementation overhead. Although using grammars is not the only way to tackle program synthesis, other approaches, especially in GP, will be reviewed and in a comparison the advantages and disadvantages will be discussed.

Further studies on the limitations and possible improvements on the grammar design approach are undertaken in Chapter 5, which analyses the computational effort required to solve program synthesis problems, and Chapter 6, which investigates limitations of the approach and how to counteract them as well as shows an example of how to further extend the proposed grammars.

Question 2: *Do different derivation tree structures defined with grammars influence the search performance?*

While grammars are widely used nowadays in the GP community [20, 21, 22, 23, 24, 25] and a range of studies involving grammars have been published [17, 26, 27, 28, 29], limited work is available on how to design grammars for a problem. In many cases, when grammars are used in tree-based GP systems, the default genetic operators are used as well. The design of

grammars influences the structure of derivation trees and in turn influences which parts and how big the subtrees are that can be manipulated by the genetic operators. In common GP setups, the trees that are going to be manipulated are binary or n-ary trees. Due to grammars the derivation trees in a G3P system, have all kinds of shapes. In Chapter 3, a study on grammars producing different derivation trees is conducted to analyse the search performance on sorting networks. This work is extended to program synthesis in Chapter 4.

Question 3: *How to define semantics and semantic measures in GP for program synthesis?*

Semantic information is the key to many improvements that have been made in GP over the last few years. Semantics has been defined in a few problem domains which all are restricted to certain types, like real numbers or boolean values. In contrast, program synthesis uses a variety of data types in combination, like integers, floats, booleans and strings. Additionally, programs can make use of control flow as well as data structures like lists, which can contain one or more different data types. No definition for semantics and semantic similarity measure exists for this problem domain. Therefore, a definition of semantics in program synthesis and appropriate similarity measures are necessary to answer this research question and will prove useful to other researchers as well. Chapter 7 is dedicated to defining semantics in program synthesis and testing different semantic measures and operators in this problem domain.

Question 4: *How can semantic information be exploited in operators to improve performance?*

Semantic information is already used in other problem domains in connection with GP and has achieved better performance than traditional approaches. For example, semantic crossover [30, 31, 32, 33, 34] and mutation [35, 36] operators have been introduced in GP which improve the performance for regression and boolean problems. As semantic information has been used successfully in other problem domains, the assumption is that similar behaviour will be observed in program synthesis as well. Chapter 7 introduces novel semantic operators based on a definition of semantics in

program synthesis given in this thesis. The new operators will be thoroughly analysed and compared to default GP operators.

1.3 Contributions

All chapters in this thesis are based on published work. The publications produced during the completion of the thesis are listed on page xx. A summary of the main contributions made to research are outlined below:

Literature review

An overview of the most relevant topics is given in Chapter 2. Section 2.1 and Section 2.2 outline the field evolutionary computation and genetic programming respectively. Section 2.3 reviews work from the program synthesis domain and relevant methods to tackle program synthesis with, especially in the area of GP. Finally, Section 2.4 surveys work on the topic of semantics.

Grammar design approach

A new approach to program synthesis is presented which uses reusable grammars that do not require to be tailored to specific problems and is competitive with the state of the art.

Grammars for general purpose programs in Python

The grammars produced for the grammar design approach to evolve general purpose programs in Python have been made available in Appendix B and extended grammars in Appendix C, as well as online for public use [37].

Insights into the general program synthesis benchmark suite

Experiments conducted in the area of program synthesis use the problems available in the general program synthesis benchmark suite, see Section 2.3.2. All experiments carried out on this set of problems have given further insight in the benchmark suite, especially because the grammar design approach was the second method tested on these problems. A discussion about specific attributes concerning the benchmark suite can be found in Chapter 5.

Definition of semantics in program synthesis

A definition and detailed description of semantics in program synthesis has

been given in Chapter 7. Based on the definition further research can be conducted, even independently of the grammar design approach.

Novel semantic operators for program synthesis

Novel semantic operators for program synthesis have been introduced and improved in Chapter 7 based on the definition of semantics in program synthesis.

Publications

Numerous publications have been produced during the completion of this thesis which are listed on page xx. All chapters in this thesis are based on published work. Each chapter states in the introduction the papers it is based upon.

1.3.1 Technical contributions

All the implementation done during the completion of this thesis has been made available online on GitHub [37], which includes plugins for a heuristic and evolutionary algorithms framework called HeuristicLab [38], which are publicly available and open source. These plugins include all the code necessary to rerun any experiment conducted for this thesis, provide support for grammar-based problems to HeuristicLab, include the grammars and an automatic grammar combiner, lexicase selection as well as all the problems from the general program synthesis benchmark suite, discussed in detail in Section 2.3.2.

Parts of the implementation have also been integrated into PonyGE2 [39] as a showcase that other systems can adopt the grammar design approach with little overhead.

1.4 Limitations

Program synthesis is a very broad field, and genetic programming offers many ways to tackle it. To this end, certain limitations had to be chosen.

Program synthesis has been tackled in this thesis with a derivation tree G3P system, similar to CFG-GP [17]. See Section 2.3.1 for more detail. The same behaviour that was achieved with the G3P system used cannot be guaranteed with linear systems as well. Parameter settings were mainly chosen from common

settings used in the literature or by following the guidelines for the benchmark problems used. No exhaustive parameter optimization took place.

As genetic programming will be used throughout the thesis, the literature review will focus on this technique also when reviewing program synthesis. Other approaches have been considered as well, but only a brief overview will be given.

Experiments conducted on program synthesis only use problems from the general program synthesis benchmark suite [1] due to the lack of better benchmark problems in GP [40, 41, 42].

1.5 Thesis Outline

The goal of this thesis is to provide a flexible grammar-based approach to tackle arbitrary program synthesis problems in arbitrary languages with a single reusable grammar design. The approach needs to be general enough to solve a variety of program synthesis problems without the need of tailoring the grammars to specific problems but offer the potential to be either extended for a broader range of problems or customized to particular tasks. To further enhance the approaches capabilities, semantic information will be studied in the domain of program synthesis and utilized in the form of novel semantic operators. The rest of the thesis is structured into four parts.

Part I includes two chapters covering the introduction and the related work. Chapter 1 contains the introduction and states the aim of the thesis as well as the research question. The contributions made by the thesis are outlined, and possible limitations are summarized. Finally, an outline of the rest of the thesis is given.

Chapter 2 reviews relevant work in the areas of genetic programming, program synthesis and semantics. Section 2.1 outlines the field of evolutionary computation and Section 2.2 explains Genetic Programming (GP) and its search process in more detail as well as grammars, which are used in Grammar-Guided Genetic Programming (G3P). Section 2.3 reviews the domain of program synthesis in general, how program synthesis has and can be applied as well as approaches also outside of GP that have tackled this specific problem domain. A focus is put upon GP approaches that were explicitly designed for program synthesis. Section 2.3 also introduces the general program synthesis benchmark suite that has been introduced in 2015 at the Genetic and Evolutionary Computation Confer-

ence (GECCO) [1]. The chapter continues with Section 2.4 which will give an overview of the research that has been conducted in the area of semantics in GP. Semantics has helped improve performance in other problem domains but has been underutilized in program synthesis. Chapter 7 will introduce semantics in the field of program synthesis.

Part II begins the experimental research. Chapter 3 is a novel study on grammars that produce the same language but produce different derivation tree structures. This study is conducted on sorting networks which provides concise grammars that can easily be analysed. The question answered with the study is, if the derivation tree structure influences the search performance. The conclusions drawn from this chapter will be of use when designing grammars for program synthesis.

Chapter 4 introduces a grammar design approach that can be used by any G3P system to tackle general program synthesis problems. The grammar design approach is tested on the general program synthesis benchmark suite and compared to the state of the art system PushGP. The grammar design approach achieves competitive results compared to PushGP while producing programs in a programming language used by practitioners.

Part III consists of three independent expansions to the grammar design approach. Chapter 5 investigates the computational effort required to solve program synthesis problems and uncovers underlying problems that prevent G3P from performing better. Suggestions on improving the success rates on problems are given, and issues with the general program synthesis benchmark suite are discussed.

Chapter 6 showcases the benefits of grammars by extending the original grammars created with the grammar design approach. Functionality that is already in PushGP and a new grammar for an additional data type are added. These changes counteract previous shortcomings in tackling certain kinds of problems.

The third expansion, in Chapter 7, introduces semantics in the program synthesis domain. A definition of semantics in program synthesis is provided, and a detailed explanation of how it can be used is given. A new semantic crossover operator based on the previous definition of semantics is created and tested. Extensive analysis of the behaviour of this operator is conducted, and improvements are suggested. Based on these suggestions novel effective semantic operators, crossover and mutation, are introduced that outperform the default genetic operators in GP as well as the previously created operator.

The last part, Part IV contains the final Chapter 8 the conclusions of the thesis, including a summary, limitations and suggestions for possible future work, as well as the appendix and bibliography.

Chapter 2

Related Work

Three topics are discussed being the main related work to this thesis. Section 2.1 outlines the field of evolutionary computation and Section 2.2 explains Genetic Programming (GP) in detail. The main application tackled, program synthesis, is reviewed in Section 2.3 along with a variety of synthesizers and a general program synthesis benchmark suite used for experiments. Finally, Section 2.4 discusses semantics and different approaches using it within GP.

2.1 Evolutionary Computation

Evolutionary Algorithms (EAs) are a set of search and optimization algorithms inspired by, but only very loosely based on, biology and the mechanism of natural selection from Darwinian evolutionary theory [43]. All EAs use a set of candidate solutions, with exceptions like (1+1) ES, which is improved iteratively. The set of candidate solutions is usually named a *population* and a single solution is called *individual*. Each iteration of improvement is called a *generation*. A general EA algorithm is depicted in Figure 2.1. First, the initial set of candidate solutions to a problem are initialised, followed by an evaluation step to determine how well a candidate solution solves a problem. Usually, the evaluation step assigns a value, named fitness value as it describes how fit a solution is, to each solution. How the fitness is evaluated depends on the problem tackled. Often input/output pairs or simulations are used. Afterwards, the stopping criterion is checked, which defines when to stop the search algorithm. Commonly used stopping criteria are if a solution is found that solves the problem correctly or well enough or a maximum

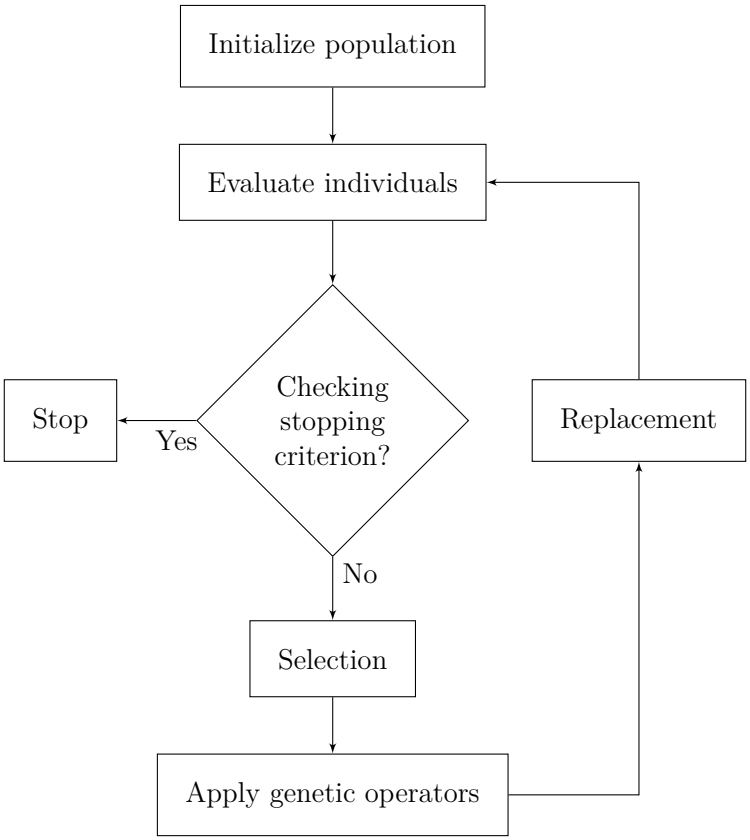


Figure 2.1: General EA cycle

number of iterations. Multiple stopping criteria can be used. If the iteration continues, selection takes place. In the selection step, fit solutions are selected that are going to be manipulated for the next cycle. Many EAs include a so-called elitism mechanism which copies the n best candidate solutions and makes them part of the subsequent iteration without changes. Genetic operators are applied to the other selected solutions. The operators are called genetic as they change the structure of the solutions and again because EAs are inspired by biology. The most common operators are a binary operator, named *crossover*, that takes two solutions and replaces parts of each of them with pieces from the other and a unary operator, named *mutation*, that takes and changes a single solution. Some EAs also include another step called *replacement* that decides which and how many of the of the previous generation and how many newly created individuals are part of the next generation. Finally, the newly generated candidate solutions are evaluated, and the stopping criterion is rechecked. This cycle is repeated until a stopping criterion is fulfilled.

EAs are a very general concept without a specified representation or set of operators that can be used. They only describe a rough search process and therefore allow to tackle any problem if an adequate representation is given.

The following four paradigms of EAs are probably the most famous and influential ones, and EAs nowadays are still often categorised in those.

Evolutionary Programming (EP) is one of the earliest works on evolutionary algorithms by Fogel [44] in the 1960s. Initially, EP used finite state machines as representation and as predictors and offsprings were created with mutation.

Evolutionary Strategy (ES) is a search technique that has been proposed and developed by Rechenberg [45] and Schwefel [46] in the 1960s and 1970s. ES usually uses a fixed length numerical vector as representation and mutation as its primary search operator. Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [47] is one well-known instance of ES.

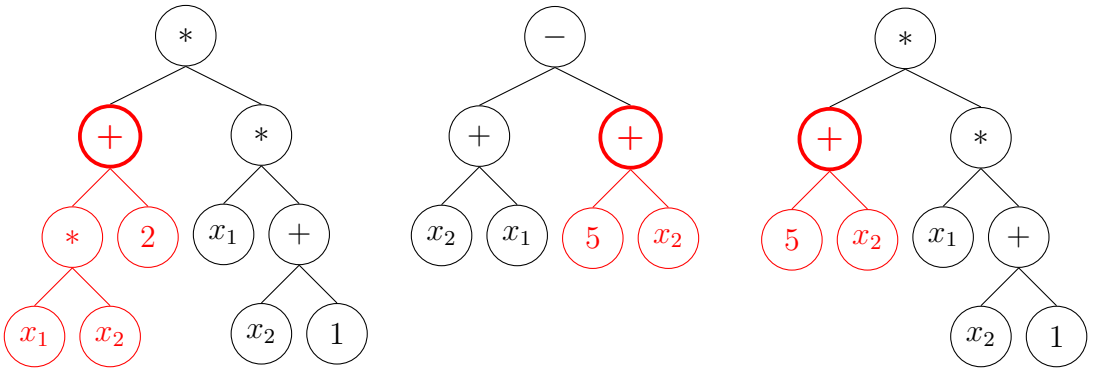
Genetic Algorithm (GA) is a famous EA invented by Holland [48] in the 1970s that evolved a population of fixed length binary vectors. The main search operators were crossover and mutation.

Genetic Programming (GP) is an EA that has become popular through work by Koza [49, 50, 51, 52]. Traditionally, a tree representation is used, where internal nodes represent functions that are executed and leaf nodes are variables and constants.

While different approaches of EAs have been invented and even been developed independently of each other in the early years, many similarities are apparent. In the early 1990s, through the emergence of conferences on EAs, the term "Evolutionary Computation" (EC) was created as a name for the field [53]. Each approach extended its repertoire of representations and operators over the years. De Jong [53] gives a unified view of evolutionary computation.

2.2 Genetic Programming

The following section focuses on Genetic Programming and describes GP in more detail as the system used throughout this thesis has most in common with GP.



(a) GP tree, parent 1 (b) GP tree, parent 2 (c) GP tree, offspring

Figure 2.2: Representation of three GP individuals and the effect of crossover. (a) represents the first parent and (b) the second parent selected for crossover. The bold red node represents the subtree selected for the crossover event. The subtree from the first parent (a) is replaced with the subtree from the second parent (b), which creates a new individual (c).

Trees will be used to represent individuals and crossover, as well as mutation, are used as search operators. Also, most of the related research is executed with GP systems as well. As most EAs, GP can be described with Figure 2.1, its main search operators are crossover and mutation and trees to represent candidate solutions are most commonly used [49, 54].

2.2.1 Representation

An example of GP trees is given in Figure 2.2. The trees consist of internal and leaf nodes. Internal nodes are functions that can be executed. All available functions to a GP system are called function set F. Leaf nodes are usually variables or constants. All available variables and constants are called terminal set T. The combination of function and terminal set is also called primitive set. The primitive set for the trees in Figure 2.2 could consist of $F = \{+, -, *\}$ and $T = \{x_1, x_2, 1, 2, 5\}$, but could contain more functions and terminals. Nodes in the tree have a property called arity, which defines how many subtrees it requires as inputs. $+$ is a binary operator and has 2-arity, although it could be defined to have any arity between 2 and infinity, by just summing up all inputs. The function *cos* has an arity of one, and terminal nodes have an arity of zero.

Two important properties in GP are *closure* [49] and *sufficiency* [54]. Closure again consists of two properties, *type consistency* and *evaluation safety*. Type con-

sistency means that any function can handle input of any type it might get. Type consistency is important as crossover and mutation change subtrees arbitrarily. Therefore, any function from the function set can get any terminal or the output of any other function as input. Evaluation safety is the second part of closure, which means that no function is allowed to fail whatever the input. An example is division by zero. Although 0 is a numeric value and division can handle numeric values, hence type consistency is given, division is not defined for having zero as divisor. But for GP to work efficiently, evaluation safety has to be given [54]. In many cases, this is avoided by defining a protected version of division that returns the dividend if the divisor is too small.

Sufficiency is a property that is difficult to achieve. It demands that it is possible to construct a solution for the problem at hand with a given function and terminal set. Even if sufficiency cannot be guaranteed, GP can still provide an approximation based on the available primitive set.

2.2.2 Initialization

After selecting the primitive set required for the problem being tackled, a set of candidate solutions, the populations, has to be initialized. The general idea behind how initialization works is straightforward. A random function or terminal is selected from the primitive set, which will become the root node of the new tree. Then, the same number of nodes have to be created with randomly selected functions and terminals as the arity of the previously created node. Afterwards, this step is repeated for every newly created node. This guarantees that functions get as many inputs as required. Theoretically, this process might never stop, if too few terminal symbols are selected. For that reason, either a tree depth or tree length limit is set, after which only terminal nodes are created. Tree depth is the maximum number of edges, connections between two nodes, it takes to reach a leaf node. Tree length is the number of nodes a tree consists of. Traditionally, tree depth is used in GP as the three main initialization methods, grow, full and ramped half-and-half [49] used a tree depth to restrict the tree sizes.

Daida et al. [55] showed that standard GP with binary trees searches rather sparse than dense trees, which is one of the reasons why a maximum tree length instead of a depth limit is used throughout this thesis. Therefore, Probabilistic Tree-Creation 2 (PTC2) [56] is used, which allows setting a tree length limit. The

other reason for a tree length limit is that grammars are used as a representation which often do not produce trees were a tree depth limit might be adequate, but that will be discussed in Section 2.2.8.

2.2.3 Fitness

During the evaluation step, each individual gets assigned a value, called fitness, that determines how well an individual solves the problem. In many GP applications, the problem is defined by a list of input and output cases. Every individual is executed on the input cases and its result is compared to the output. While in some cases it is adequate to capture how many cases have been solved correctly, in other scenarios it can be useful to use the information of the difference of the result the individual produces and what the correct output was. The function used for the evaluation is also called fitness function. Other forms of evaluation include for example simulations or even humans judging the output of a candidate solution.

After the fitness evaluation, usually, the stopping criteria are checked, which in most cases consist of if a maximum number of iterations (generations) has been used up, and if a solution has been found that solves all cases correctly. If one of these criteria is fulfilled, the GP cycle, also called run, is stopped. Other stopping criteria can be used as well.

2.2.4 Selection

Selection is the process of deciding which individuals are going to be part of the next generation after undergoing changes through the search operators. Individuals can be selected multiple times during this process. Most selection operators used in GP are stochastic, but favour individuals with a higher fitness value. The most popular [57] selection operator is tournament selection [58]. Tournament selection uses tournaments to decide which individuals are going to be selected. n individuals, which is the tournament size and can be set by the practitioner, are randomly selected from the whole population to be part of a tournament. The best individual from a tournament is selected to be part of the next generation. Many other selection operators exist which are also used in GAs, like fitness proportionate selection or rank selection [59]. Another selection operator that should

Algorithm 2.1 A single selection event with lexicase selection

```
set candidates to be the whole population
randomize training cases
repeat
  candidates ← candidates that are most successful on the first training case
  remove first training case
until candidates only contains a single candidate or no more cases
if candidates only contains a single candidate then
  return candidate
end if
return a random candidate from candidates
```

be mentioned, apart from tournament selection, which is going to be used in two of the experimental chapters, is lexicase selection.

Lexicase selection

Lexicase selection [60, 61] is a recent selection operator compared to the other ones named above. The reason for describing it in more detail is for one that it is rather new, and that it was shown to be more successful than any other selection operator in the program synthesis domain, which is why it is used in almost all chapters of this thesis.

In contrary to many other selection operators, lexicase selection does not operate on a single aggregated fitness value given to an individual. Lexicase selection uses all fitness cases achieved on the input/output cases tested during evaluation. A single selection event executed by lexicase selection is described in pseudocode in Algorithm 2.1. Training cases are the input/output cases an individual is evaluated on.

First, the whole population is set to be candidates to be selected, and the training cases are randomized. Then, an iteration starts, where the first step is to check how well the current candidates did on the first training case. The current candidates get replaced with only those who achieved the best result on the first training case. Afterwards, the training case is removed for this selection event. If only one candidate is left, it is the one that has been selected during this selection event. Otherwise, the iteration continues and the next training case is checked. If all training cases have been checked and there are still more than one candidates left, then a random one is selected.

The name lexicase selection originates from "lexicographic ordering" as is done with strings as the first training case selected has the largest effect and the next one only matters for ordering within the smaller set [60]. Lexicase selection has been proven to perform well on multimodal [60] and "uncompromising" [61] problems. Uncompromising problems are ones that require the candidate solution to perform optimally on all cases. La Cava et al. [62] introduced ϵ-lexicase an adaption of lexicase selection for the continuous space, which uses a threshold for choosing if an individual solved a training case better than another.

2.2.5 Crossover

Crossover is one of the search operators applied by GP to create individuals for the next generation. Crossover is a binary operator which takes two individuals to combine them in some way to create the offspring. Figure 2.2 shows three GP trees. Lets assume that the first two, Figure 2.2a and Figure 2.2b, have been selected for crossover. Then a random crossover point is selected in both of them, which is represented by the bold red node in both trees. The subtree in red from the first parent, Figure 2.2a, gets replaced with the subtree in red from the second parent, Figure 2.2b. This process creates a new individual shown in Figure 2.2c. Many different approaches for selecting crossover points and different crossover operators exist [54].

Koza [49] used an adapted version of the explained procedure that favours internal nodes as crossover points, choosing internal nodes with a 90% probability and leaf nodes with 10%. The reason is that due to the arity of function nodes, which often is two or higher, more leaf nodes than internal nodes are created. This leads to many crossover events only exchanging single nodes or small subtrees, if choosing uniform randomly from all nodes. This crossover operator is often called Koza-style crossover and is used in the experiments in this thesis unless otherwise specified.

2.2.6 Mutation

Mutation is the second search operator used in GP. In contrary to crossover, mutation is a unary operator. Mutation takes a single individual and changes some part of the tree, for example by replacing a subtree with a randomly created one. Let's assume the tree in Figure 2.2a has been selected for mutation and the

bold red node has been selected for the mutation event. Then, its subtree marked in red is a randomly created tree that has been added.

It is often argued about if mutation is necessary [54] and Koza et al. [63] suggest to only mutate a small number of individuals. Luke et al. [64] showed that it might depend on the problem and the details of the GP system how well crossover and mutation perform.

2.2.7 GP Summary

So far, this section gave an overview of a traditional GP system. Many advancements have been made since GP has been introduced. A summary of the field can be found in "A field guide to genetic programming" [54], but also many open issues still have to be addressed [65]. Nowadays many different variants and implementations of GP exist. Some of which will be discussed in Section 2.3.1. As many of them also use different representations, Section 2.2.8 is going to explain grammars with a focus on Context-Free Grammars. Knowledge about grammars is crucial when talking about Grammar-Guided Genetic Programming (G3P), which is the GP variant used for the experiments. Due to grammars, the property of closure is not as important anymore, as the structure of the output is defined in the grammars and can restrict the input of functions. Additionally, bias can be added in grammars, e.g. in the form of expert knowledge. G3P will be discussed in the context of program synthesis in Section 2.3.1.

2.2.8 Grammars

Although there is much theory on grammars, only the essentials for GP will be covered here. A grammar G can be defined with a 4-tuple $G = (N, \Sigma, P, S)$.

N is the set of non-terminal symbols. A non-terminal symbol is a symbol that can be replaced with other symbols.

Σ is the set of terminal symbols, which are symbols that appear in the final output string or sentence. N and Σ are disjoint. $N \cap \Sigma = \{\}$

P is the set of production rules that define which symbols can be used to replace other symbols.

S defines the start symbol $S \in N$.

```
<expr> ::= <expr> <op> <expr>
         | (<expr>)
         | <pre_op> (<expr>)
         | <var>
<op> ::= + | - | *
<pre_op> ::= sin | cos | exp | log
<var> ::= x1 | x2 | 0 | 1 | 2
```

Figure 2.3: Context free grammar

Figure 2.3 shows an example of a grammar in Backus–Naur Form (BNF) that can be used for regression problems. The 4-tuple (N, Σ, P, S) is often given implicitly in BNF. The start symbol S is usually the first symbol in the grammar. In case of Figure 2.3, $S =$ `<expr>`. Non-terminal symbols are all the symbols that start and end with angle brackets. $N =$ {`<expr>, <op>, <pre_op>, <var>`}. All other symbols are terminal symbols. $\Sigma =$ { `(, ), +, -, *, sin, cos, exp, log, x1, x2, 0, 1, 2`}. Finally, P are all the rules shown in Figure 2.3. Every rule has a left-hand and right-hand side, which is separated by "`::=`" symbol. This grammar is actually a Context-Free Grammar (CFG), which is most commonly used in GP [17, 18, 66]. Therefore, the left-hand side is always a single non-terminal symbol. The right-hand side represent the possible productions the left-hand side can be replaced with. The productions are separated by the "`|`" symbol.

Replacing a symbol with its productions is called a derivation. Representing the whole tree of derivations is called a derivation tree as shown in Figure 2.4, which is often used in GP variants that use grammars [17]. Figure 2.4 shows an example of a derivation tree of the grammar from Figure 2.3. The root node of the derivation tree is the start symbol S, which is the replaced with a production `<expr> <op> <expr>`. The derivations continue until all non-terminal symbols are replaced. The resulting string is called a sentence, which one gets by concatenating all leaf nodes from the derivation tree. The sentence represented by Figure 2.4 is `x2 + sin(x1)`. All possible sentences that can be derived from G are called a language, which is denoted $L(G)$.

The grammar in Figure 2.3 has one more property that should be noted, which is that it is a recursive grammar. A recursive grammar has productions that replace a symbol, and after a number of derivation steps create a previously

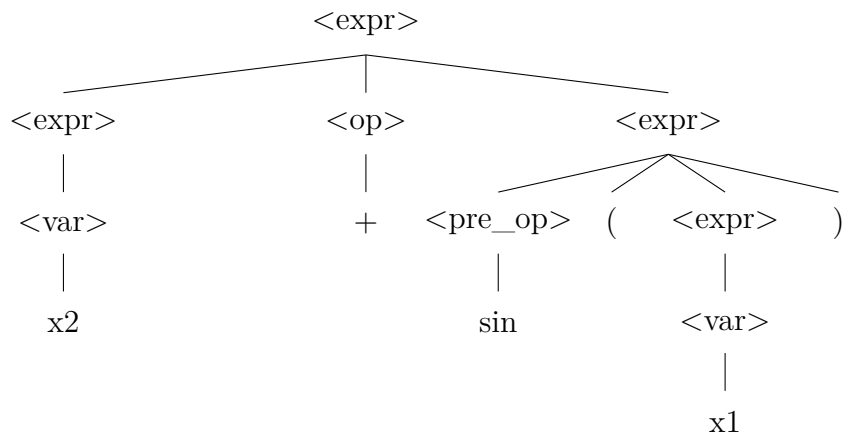


Figure 2.4: Example derivation tree of the grammar from Figure 2.3.

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

Figure 2.5: Variable length GA like representation for binary vector in BNF.

replaced symbol again. The grammar used as example here can replace `<expr>` with `<expr> <op> <expr>`, which leads to a recursion which only stops if replaced with `<var>`. Recursive grammars are usually more interesting in the context of GP, as non-recursive grammars may be enumerated and therefore require no search.

GA and GP Grammars

Whigam [17] showed in his PhD Thesis that grammars can also be used to represent GA and GP representations. Figure 2.5 shows a grammar for a variable length binary vector representation often used in a GA. Fixed length vector representations are not more complicated but require more rules. Figure 2.6 depicts a grammar for a fixed length binary vector.

```
<stringpart0> ::= <stringpart1><bit>
<stringpart1> ::= <stringpart2><bit>
...
<stringpartN> ::= <bit>
<bit> ::= 0 | 1
```

Figure 2.6: Fixed length GA representation for binary vector in BNF.

```
<tree> ::= f1 <tree> ... <tree> | f2 <tree> ... <tree> | ...
         | fx <tree> ... <tree>
         | t1 | t2 | ... | ty
```

Figure 2.7: General standard GP grammar in BNF.

GP trees can be represented with the grammar shown in Figure 2.7. $\texttt{f}_1$ to $\texttt{f}_x$ can be replaced with the function set used for a problem in GP and the arity of each function specifies the number of `<tree>` non-terminal symbols following. $\texttt{t}_1$ to $\texttt{t}_y$ can be replaced with the required terminal set from GP. These examples show that grammars are a fairly flexible method to represent problems.

In contrary to GP trees, whose trees can be well balanced, although that is often not the case [55], grammars can describe derivation trees that produce other shapes. For example, a derivation tree for Figure 2.5 will be more similar to a list than a tree. This is a reason why instead of a maximum tree depth limit, which is often used in GP, a maximum tree length limit will be used, as also stated in Section 2.2.2. Chapter 3 will investigate derivation tree shapes produced by different grammars that produce the same language.

The next section will introduce program synthesis and several GP systems which can be used to tackle this problem. Afterwards, Section 2.3.1 will further discuss Grammar-Guided Genetic Programming (G3P) systems in the context of program synthesis.

2.3 Program Synthesis

Program synthesis is a research area that due to its importance is pursued by many different fields. The original idea of program synthesis already existed in the 1950s [67, 68]. One of the earliest works considered program synthesis was done by Church [67] although on circuits rather than programs. Even when GP was popularized by Koza [49] to create programs, evolving circuits was a problem often tackled. A lot of work in GP was inspired by the idea of: "How can computers be made to do what needs to be done, without being told exactly how to do it?" by Koza [69] paraphrasing Samuel Arthur [68] which describes the field of machine learning. Gulwani [70] gives a modern definition of program synthesis:

> "Program Synthesis is the task of discovering an executable program from user intent expressed in the form of some constraints."

In some fields, like GP, all outputs created by a system are called programs. Although technically correct, many outputs are merely boolean expressions or arithmetic formulas. The focus in this thesis lies in synthesizing general purpose programs from scratch that can include memory and control flow, like iterations and recursion.

Program synthesis is a type of automatic programming, a term that has often been used in the GP community [71, 18]. Automatic programming is a very broad and unspecified term that includes e.g. the generation of a program by compilers and has been called "a euphemism for programming in a higher-level language than was then available to the programmer" [72]. While compilers are translators that take a program written in a structured language and do syntactical translations usually to a lower level language, synthesizers can take all kinds of different inputs to search for an executable program. Three dimensions can typically describe synthesizers: user intent, search space and search technique [70].

The user intent describes the task the program should be executing. Using natural language to describe a task would be most beneficial, but has the disadvantage of ambiguity [73]. Another form is logical expressions [74], but those require knowledge of logic, and it might be difficult to describe the task correctly with them. If the task at hand is to optimize a program or discover new algorithms, a complete program can be used to describe the user intent. Instead of a whole program, even a trace can be used [75]. A trace is a step-by-step instruction of how to transform a given input to a corresponding output. Lastly, input/output examples can represent the user intent, which provides users with a simple form of demonstrating a task. Input/output examples will be used in the experiments throughout the thesis. It should be mentioned that when using input/output examples, that a correct program has only be proven to be correct on the examples given, which is why a test set of input/output examples is used after learning to confirm a programs ability to generalize.

The search space defines all programs the synthesizer is searching over. Similar to GP, the search space should be sufficient, which means that the search space contains the correct program, but at the same time should not be too large, so that the search is probably inefficient. The search space can be restricted by the

operators used and the control structures allowed to search only for specific types of programs. The GP equivalent is the restriction of the function and terminal set. Common ways to set the search space are logic representations [76, 77] or various kinds of grammars [78, 79, 13].

Lastly, synthesizers vary in the search technique that is applied. Search techniques include but are not limited to exhaustive search [9, 80], version space algebra [11, 6, 81], logical reasoning based techniques [82, 74] as well as evolutionary algorithms. A range of different genetic programming systems are discussed in Section 2.3.1.

MagicHaskeller [9, 80] is a synthesizer that creates functional Haskell programs relying on exhaustive search and can learn from a few examples. It uses a Monte-Carlo search to remove semantically equivalent programs [83] to speed up the search.

Version space strategy was described by Mitchell [84] to constrain a set of, initially, all hypotheses and restricts it by removing a hypothesis if it is inconsistent with a new example. Based on version spaces, Lau et al. [11, 6] introduced version space algebra. A version space contains a set of simple functions, which can be combined to form complex functions. Lau et al. [6] have used that approach to create Python programs by learning from traces. Flash Fill [81] is a synthesizer for string manipulation that is based on version space algebra, learns from examples, and its functionality is available in Microsoft Excel.

Logical reasoning based techniques [82, 74] reduce the program synthesis task to logical constraints that can be solved by off-the-shelf Satisfiability (SAT) and Satisfiability Modulo Theory (SMT) solvers. SKETCH [82] uses partial complete programs, so that some control flow structure might already be given, and lets a SAT solver fill in the details. SKETCH's own syntax can be used to restrict what can be filled in a "hole" left to be filled by a SAT solver.

The following section gives an overview of relevant genetic programming systems in the domain of program synthesis, followed by Section 2.3.2 which introduces a benchmark suite consisting of program synthesis problems that are going to be tackled in this thesis.

2.3.1 Program Synthesis in Genetic Programming

This section reviews a variety of GP systems that have been used to tackle program synthesis tasks. Many more exist, but the ones discussed have been selected due to their capabilities of tackling general purpose program synthesis problems.

Grammar-Guided Genetic Programming

Many different representations have been used with GP. While the most common one is still trees, grammars are widely used as well. Especially, Context-Free Grammar Genetic Programming (CFG-GP) [17] and Grammatical Evolution (GE) [18, 66], two versions of so-called Grammar-Guided Genetic Programming (G3P), which are responsible for the success of grammars in GP.

G3P systems, like Grammatical Evolution [18] or CFG-GP [17], have been used to evolve code for program synthesis in arbitrary languages. Grammars are very flexible and can be used to represent the search space of a wide range of problems, evolving music [20], creating truss design [21], optimising pylon structures [22], evolving aircraft models [23], controlling femtocell network coverage [24] and program synthesis [13]. A G3P system does not require any additional code to evolve a solution, except a grammar and a fitness function that can evaluate individuals created by the grammar. The code that is evolved in G3P can be in an arbitrary programming language, which is defined by the grammar.

CFG-GP is a variant of tree-based G3P as individuals are represented as derivation trees generated from the grammar. Genetic operators in CFG-GP adhere to the rules a grammar describes and only exchange subtrees of an individual, with a subtree of the same non-terminal symbol as the root node to always create a valid individual.

Grammatical Evolution is a linearised G3P variant. Instead of using a derivation tree, GE uses a variable length list of integers, originally a binary vector [85]. GE has a mapping process that takes one integer after another from the individual, to replace a non-terminal symbol with a production depending on that integer. The mapping process starts with the start symbol and ends when all non-terminal symbols have been mapped or when no more integers are available. Therefore, it is possible that an individual is invalid, as the output produced by the mapping contains non-terminal symbols. The mapping process implicitly produces a derivation tree. GE is one of the most widely applied GP methods [86].

Many different variants of G3P systems have emerged using different types of grammars, context-sensitive grammars [87], attribute grammars [88] or logic grammars[89, 90] as well as different mapping processes for GE [91, 92, 93]. A survey of G3P is available by McKay et al. [86] and by Brabazon et al. [94, Part V].

Even though G3P systems are perfect candidates for tackling problems in the program synthesis domain, so far they have been hugely underutilised. No general concept for grammars exists that can tackle arbitrary program synthesis problems. Every grammar that has been used to evolve programs has been tailored to a single specific problem and cannot be reused without changes being made.

PushGP

PushGP [95] evolves programs in the Push programming language that was solely developed for research in the field of evolutionary computation. Implementations of Push and PushGP are available in many programming languages including Python and C#. The reference implementation is in Clojure.

In contrast to many common programming languages, Push uses a stack-based execution architecture. It has a stack per data type and additionally one for instructions that are going to be executed. Due to this architecture, Push does not need variables as data is stored on the corresponding stack and the execution of code can be manipulated at runtime as the stack containing instructions can be changed as well.

In PushGP there is no need to declare the number of variables beforehand, as data is stored on stacks. All available instructions to PushGP are implemented in the corresponding programming language in a protected way which avoids runtime exceptions. By using wrapper functions, functionality and libraries of the programming language, PushGP is run on, can be made available during evolution.

As Push is a language not used outside the research community, Push code cannot directly be integrated into existing software. The Push interpreter is required as well, which adds overhead. Push code can be run in different programming languages, given an interpreter, but only if the core Push functionality is used. This means wrapper functions to use libraries from the programming language PushGP was run on, cannot be executed on any other Push interpreter.

Strongly Formed Genetic Programming

Strongly Formed Genetic Programming (SFGP) [96] is an extension of Strongly Typed Genetic Programming (STGP) [12]. While standard GP operates on the premise that a function in a node must be able to handle any input from its subtrees, STGP changed the behaviour of its genetic operators (initialisation, crossover and mutation) to adhere the types, specifically node-types and data-types, available in STGP. Data-types are similar to data-types in programming languages, like integers or strings. Node-types, such as `Statement`, `CodeBlock` or `Loop`, are used to restrict the shape and structure of a program. Every function in STGP outputs a specific type and requires specific types as input. Its genetic operators are not allowed to create or change any (sub-)trees that are not coherent with the required types. Therefore, evolved individuals are type consistent, which is important for program synthesis as many different data types may be used.

SFGP extends STGP by adding additional nodes that allow the generation of generic code, like nodes for control flow (e.g.`CodeBlock` or `ForLoop`) and variables (`Variable`). Every node needs to be implemented within the SFGP system. Many aspects in SFGP have to be predefined, like a fixed number of statements/subtrees per code block, so they do not have a variable length and the number of variables as they are part of the terminals. SFGP only evolves programs in its own structure, which can be viewed as pseudocode, which needs to be translated to be used in a real-world program.

Although SFGP is an extension to STGP that allows evolving code for program synthesis problems, it exhibits many limitations. The evolved code cannot be used in a real-world program. Any new functionality has to be implemented in the SFGP system and cannot be transferred to any other system, unlike grammars, which can be used by any G3P system.

GPPS

The Genetic Programming Problem Solver (GPPS) by Koza et al. [51] is a system that was designed as "a general-purpose method for automatically creating computer programs to solve a problem." GPPS consists of an input and output vector to use various inputs and outputs, an indexed memory and is capable of using subroutines and loops. In the version 2.0, recursion was included as well. A variety of symbols are available which include usual GP symbols, like arithmetic

and logical operations, but also symbols to read and write to memory and conditional branching operators. The idea was that the same terminal and function set could be used for each problem tackled. GPPS has been used to solve a variety of problems in the boolean and regression domain as well as to control a robot and design a minimal sorting network.

Similar to most GP systems, GPPS relies only on numerical values. Boolean values are represented as numbers, and logical operations return numerical values representing the boolean value, either +1.0 or –1.0. GPPS's use in program synthesis is limited as char and string cannot be used as well as data structures. The length of the maximal length of the input and output vector has to be predetermined. As the output of GPPS is a program with its own defined function set, it can only be run with its own interpreter or the program can be used as pseudocode and reimplemented in a programming language used by practitioners.

Although most of the problems tackled with GPPS have been toy problems, GPPS has evolved a sorting network with fewer steps as a previously human-designed one [51].

Genetic Improvement

The field of Search-Based Software Engineering (SBSE) [97, 98, 99] aims to tackle software engineering with search-based optimization techniques. The most used search-based optimization techniques in SBSE are evolutionary algorithms [98]. In the GP community and closely related to SBSE, Genetic Improvement (GI) [100] emerged, which uses existing code to improve a program, e.g. by enhancing non-functional requirements or by fixing bugs.

The system GISMOE [14] automatically creates a BNF grammar from existing code. Terminal symbols in the grammar are complete lines of that code. The block structure of the code cannot be changed in GISMOE, e.g. opening and closing brackets in C++, but the contents of these blocks can be. GISOME is a GI system, and therefore its main goal is to improve existing code. It does not create code from scratch, but reuses the existing code of a program and adapts it. Inspired by the work done with GISMOE, Haraldsson et al. [4, 5] developed their own GI system within a real-world application, which as able to solve 22 bugs within the first six months of deployment. The system was operating outside

office hours, parsing error logs, reproducing errors, generating test data and fixing the errors.

Abstract Syntax Trees (ASTs) are another representation that can be used for program synthesis instead of grammars. GenProg [101] or Gen-O-Fix [102] are two such systems that utilise ASTs to synthesize code by reusing existing code. As the name already suggests, ASTs represent the syntax of code in a tree structure, similar to a derivation tree of a grammar. The tree can then be modified in a manner like trees in SFGP. Crossover and mutation are only allowed to exchange nodes of the same type to keep the chance of compilation errors low.

Using ASTs and the grammar representation used by GISMOE have the disadvantage that new individuals might be invalid in the sense that they might not compile due to syntactical errors. Nevertheless, both approaches have been used in real-world applications, GISMOE to improve the performance of programs [14], GenProg and Haraldsson's et al. GI system to fix bugs [101, 5]. GI systems are specialised to modify existing code, which is a different kind of problem compared to the systems presented so far.

2.3.2 General Program Synthesis Benchmark Suite

The general program Synthesis benchmark suite was introduced by Helmuth and Spector [1, 103] as a well defined set for comparing different approaches in the program synthesis domain due to researches highlighting the need for better benchmarks [40, 41, 42]. The benchmark suite consists of 29 problems which have been selected from iJava [104] and IntroClass [105]. iJava is an interactive computer science textbook to learn Java. IntroClass is a benchmark suite for program repair of introductory course programs written by students, although in case of the benchmark suite the purpose is to use these problems to evolve programs.

For each problem in the benchmark suite, a description of the problem itself as well as how to generate the training and test set is given. When the benchmark suite was introduced, it was tackled with PushGP. Therefore, the Push instruction set for each problem was listed, as shown in Table 2.1. Parameter settings for PushGP have also been given, but apart from the number of generations and population size, these settings might not be useful for other GP systems.

Table 2.1: Push instruction set used per problem as well as the number of training and test cases.

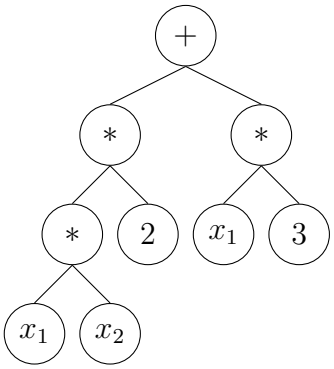
Problem	exec	integer	float	boolean	char	string	vector of integer	vector of floats	vector of strings	print	file input	Training	Test
Checksum	x	x		x	x	x				x		100	1000
Collatz Number	x	x	x	x								200	2000
Compare String Lengths	x	x		x		x						100	1000
Count Odds	x	x		x			x					200	2000
Digits	x	x		x	x	x				x		100	1000
Double Letters	x	x		x	x	x				x		100	1000
Even Square	x	x		x						x		100	1000
For Loop Index	x	x		x						x		100	1000
Grade	x	x		x		x				x		200	2000
Last Index of Zero	x	x		x			x					150	1000
Median	x	x		x						x		100	1000
Mirror Image	x	x		x			x					100	1000
Negative To Zero	x	x		x			x					200	2000
Number IO		x	x							x		25	1000
Pig Latin	x	x		x	x	x				x		200	1000
Replace Space w. Newline	x	x		x	x	x				x		100	1000
Scrabble Score	x	x		x	x	x	x					200	1000
Small Or Large	x	x		x		x				x		100	1000
Smallest	x	x		x						x		100	1000
String Differences	x	x		x	x	x				x		200	2000
String Lengths Backwards	x	x		x		x			x	x		100	1000
Sum of Squares	x	x		x								50	50
Super Anagrams	x	x		x	x	x						200	2000
Syllables	x	x		x	x	x				x		100	1000
Vector Average	x	x	x					x				100	1000
Vectors Summed	x	x					x					150	1500
Wallis Pi	x	x	x	x								150	50
Word Stat	x	x	x	x	x	x	x	x	x	x	x	100	1000
X-Word Lines	x	x		x	x	x				x		150	2000

Push contains instructions for printing and file operations. While it can be helpful to have such instructions, simply returning values instead of printing them is often preferable in programming when developing a method. The methods evolved in this thesis aim to only use input and output values and involve no printing, though they theoretically could be added in an existing software system. Therefore, instead of printing values, they are returned. This is a technical detail, which changes the problems, although they are still similar. The original description of the problems is available in form of a conference paper [1] and a technical report [103]. A description of each problem of the general program synthesis benchmark suite with returning instead of printing values is available in Appendix A as well as the corresponding fitness functions.

2.4 Semantics

For a long time in GP and many evolutionary algorithms, operators have been designed to manipulate the internal representation without regarding how it will affect the behaviour. In tree-based GP, common operators manipulate the tree structure by randomly swapping subtrees without checking before or after the effect of a change. The benefit of such operators is that they can be applied in any problem domain as they rely on the internal representation of the system used.

Since then a lot of research has happened in the field of semantics. A commonly accepted definition of semantics in the GP community is that semantics refers to "the behavior of a program, once it is executed on a set of data" [16]. Depending on the problem domain "semantics" and "program" can mean something different. In the regression domain, a program usually is a formula and semantics is a vector of real values, while in the boolean domain a program is a boolean expression and the semantics is a vector of booleans. Figure 2.8 illustrates the differences between the internal representation (2.8a), the program (2.8b) and the semantics (2.8c) in the regression domain. Figure 2.8a depicts a randomly generated GP tree, and the program or formula represented by that tree is shown in Figure 2.8b. The last Figure 2.8c contains the semantic produced by the formula when executed on the data from the first two columns. The data can be either a subset of or the whole training data or randomly sampled data. The last column contains the semantics of the whole tree, which is a vector of real values. Many use cases for semantics



(a) GP syntax tree representation or genotype

$$x_1 * x_2 * 2 + x_1 * 3$$

(b) "Program" or phenotype

Data		Subtree semantics			Semantics
x_1	x_2	$x_1 * x_2$	$x_1 * x_2 * 2$	$x_1 * 3$	output
1	0.1	0.1	0.2	3	3.2
2	0.2	0.4	0.8	6	6.8
3	0.3	0.9	1.8	9	10.8

(c) Semantic output of the tree given some data.

Figure 2.8: The three figures represent the internal GP representation genetic operators manipulate (a), the syntactical representation of the tree (b) and its semantics (c). The semantics in (c) in the last column is calculated depending on the data from the first two columns. The columns in between contain the semantics of two subtrees.

do not require the output of the whole tree, but of subtrees which are depicted in the columns between the data and the semantics of the whole tree. The semantics of a tree for the training data is calculated by GP to determine the fitness of a candidate solution. Therefore, no computational overhead is required to obtain this information. This is also true in many cases for obtaining the semantics of subtrees.

An aspect about semantics that should be pointed out is that there is a difference between syntactical and semantical identical. The following formulae are syntactically different but semantically identical as they produce the same output if the same input is given:

$$f_1(x, y) = x * y * 2 + 3 \tag{2.1}$$

$$f_2(x, y) = x * (y + y) + 2 + 1 \tag{2.2}$$

This differentiation between syntax and semantics is essential. Individuals with the same semantics have the same fitness but can have different syntax. Therefore their internal representation is different and so different syntax can be contributed to the next generation by semantically identical individuals. This is not the case for syntactically identical individuals.

As this explanation shows, semantics is different depending on the problem domain. Hence, how semantics is used requires adaptation for each problem domain as well. Most research on semantics to date has been carried out in the boolean [30, 31, 35] and regression domain [32, 33, 34, 36]. Both domains benefit from using only a single data type, boolean and real values respectively, and the semantics can be defined as a single vector of that data type. Chapter 7 will address semantics in general program synthesis, which requires multiple different data types and multiple vectors for semantics.

Two important properties of semantics in general that contribute to performance improvements are *semantic diversity* and *semantic locality* [34, 16]. A high semantic diversity is necessary for covering the search space and has been shown to be more helpful than structural diversity [16]. Semantic locality means a small change in a program corresponds to a small change in its semantics and therefore fitness. While keeping up semantic diversity is necessary, semantic locality is essential for the search performance [34].

The next two sections review a range of semantic operators. Section 2.4.1 covers GP operators, crossover, mutation as well as selection, that have made use of semantic information by promoting semantic diversity, semantic locality or both. Section 2.4.2 gives an overview of a more direct approach of using semantic information but also shows its limitations. Apart from the operators presented in the next sections, other ways to use semantic information exist by using formal methods or grammars which are reviewed in [106], but will not be covered as they are not relevant to this thesis. Although grammars are a central topic in this thesis the use of semantics in grammars is focused on checking semantic validity of solutions by using attribute grammars [88] or logic grammars[89, 90], which might be difficult to extend to program synthesis.

2.4.1 Semantic operators

In the boolean domain, Beadle et al. [30] proposed a Semantically Driven Crossover (SDC) that used Reduced Ordered Binary Decision Diagrams (ROBDDs) to compare children to their parents. Boolean expressions that produce the same ROBDD will produce the same semantic output. By using RODBBs, it is not necessary to evaluate the fitness of a boolean expression to check if two expressions are semantically equivalent. SDC reduced parents and children to their ROBDD

form and test for semantic equivalence. If a child was semantically equivalent to its parent, it was not added to the next generation and the process of crossover was repeated. It was shown that SDC was a statistically significant improvement over standard crossover. The work was extended to Semantically Driven Mutation (SDM) that operated in a similar fashion, but instead of selecting a subtree from a second parent, a randomly created one replaces an existing subtree. Again the mutated program is only added to the population if its ROBDD is not semantically equivalent.

Work by McPhee et al. [31] shows the importance of using semantics. They investigated the behaviour of crossover in the boolean domain by enumerating all possible inputs. In the experiments conducted, they showed that typically 75% of crossover events happening do not affect the semantics of the whole expression. The bigger the tree size, the more likely it was that a crossover event is not having an effect.

Nguyen et al. did extensive research on semantics in the regression domain [32, 36, 33, 106, 34]. Similar to [30] a crossover operator was introduced [32] that checked semantics to produce children that are semantically different from their parents. Contrary to boolean expressions, checking the equivalence of arithmetic expressions is NP-hard [107] and enumerating the space of real values is impossible. Parents and children were measured on randomly sampled points from the domain of the problem to check for semantic equivalence. A semantic sensitivities parameter was used to decide if the expressions were too similar. This idea was extended to two mutation operators, Semantic Aware Mutation (SAM) and Semantic Similarity-based Mutation (SSM), in [36]. While SAM works similar to the crossover in [32], SSM checks for semantic similarity. The check for semantic similarity is similar to the check for semantic equivalence, but a lower and upper bound is used to make sure a subtree is not equivalent, which would not lead to a change in the semantics of the expression, and not too different. Hence, SSM was not only promoting semantic diversity by changing the semantics, but also semantic locality by not changing it too much to affect the semantics of the expression and therefore its fitness only slightly. As semantic similarity is more difficult to achieve than semantic difference as the difference has to be between an upper and lower bound compared to only being higher than a lower bound, SSM could use multiple trials to find and generate subtrees that show semantic similarity. If a maximum number of trials was reached, standard mutation was executed. In the

experimental results, SSM was outperforming SAM and standard mutation, while SAM achieved similar results to standard mutation.

In a next step, the improvements of SSM were brought back to a new crossover operator, Semantic Similarity-based Crossover (SSC) [33], that outperformed the previous semantic crossover and many syntax-based crossover operators. In [34], Nguyen et al. introduced a Self-Adaptive Successful Execution (SASE) approach to adapt the upper and lower bounds for SSC and another semantic crossover, Most Semantically Similar Crossover (MSSC). The benefit of MSSC is that no upper bound was required for the semantic similarity. The semantic similarity for a number of crossover events was calculated. The crossover event that achieved the most semantically similar result was executed in the end. It should be noted that for all the semantic mutation and semantic crossover operators in the regression domain no additional *fitness* evaluation was executed, although computational overhead arises due to calculating the semantic similarity between subtrees. MSSC has shown to improve the performance of GP on benchmark problems as well as two real-world problems [34], which is attributed to the higher semantic locality of MSSC.

Selection

Another operator that has benefited from semantics is selection. Most selection operators, especially from the early years of evolutionary algorithms are solely based on fitness. On the one hand, due to only using fitness, selection operators can be used independently of the representation, which makes them even more flexible than crossover or mutation operators which are based on syntax. On the other hand, semantic selection operators have been introduced which outperform traditional ones. Two operators that use semantics during the selection process are Semantics in Selection (SiS) [108] and Semantic-Clustering Selection (SCS) [109]. Both selection operators always select two individuals as the next operator applied is crossover, which combines parts of two selected individuals. SiS uses standard tournament selection to choose the first parent for crossover. The second parent is also selected with a tournament, but the parent is also checked if it is semantically different from the first one to promote semantic diversity. SCS uses a different approach, by clustering individuals based on their semantics. Again the first parent is selected by standard tournament selection from the whole population. The

second parent is then selected with tournament selection from the same cluster as the first one to be semantically similar to the first one. The idea is to promote semantic locality within the cluster and having semantic diversity by having multiple clusters.

Apart from using the semantic output, selection techniques have been invented that are based on the error vector which contains the fitness of every training cases. This approach is more fine-grained than using an aggregated fitness value as in traditional methods and is more informative than using the semantic vector as the error vector estimates how well the individual does on every training case. One such operator, lexicase selection which is very successful in the program synthesis domain, has already been discussed in Section 2.2.4, another one is Semantic Tournament Selection (STS) [110]. STS promotes semantic diversity by using a statistical test instead of solely relying on fitness. An individual is only better than another one in a tournament if it is significantly different and more fit than another one. The benefit of relying on the error vector instead of the semantic vector is that the selection mechanism can be applied on any problem that uses multiple training cases instead of having to adapt a semantic measure for every problem domain.

2.4.2 Geometric Semantic GP

A more direct approach of using semantics is to create crossover and mutation operators that induce a unimodal fitness landscape. Such operators are called geometric due to exploiting the geometric properties of the search space. Based on work from Krawiec et al. [111, 112], Moraglio et al. introduced Geometric Semantic Genetic Programming (GSGP) [113]. Geometric semantic crossover and mutation operators have been proposed. Due to the induced unimodal landscape, the crossover operator cannot produce a child that is worse than either of its parents. GSGP performed better than standard GP in the experiments executed in [113]. The geometric semantic mutation was even used on its own in a hill climber, which was able to outperform standard GP.

A major problem in GSGP is that solutions rapidly grow in size. The reason is that crossover combines the whole tree structures of the parents for a new child and mutation adds additional nodes to the tree before the root node. Hence, crossover leads to exponential and mutation to linear growth. The growth problem

has only been addressed with implementation tricks [114] which do not actually reduce the size or by post-simplification-steps [115] that produces solutions that are still orders of magnitude bigger than what standard GP produces.

GSGP also has a range of other issues. GSGP can only be applied to a restricted number of problem domains for which geometric operators can be created, like boolean and regression problems. GSGP produces overfit solutions and has been outperformed on a broad set of benchmarks by other GP approaches [116]. Pawlak [117] criticised that "Geometric Semantic Genetic Programming Is Overkill". Even though the theory behind GSGP is correct, GSGP only produces a linear combination of random parts. In the same paper [117], linear combinations of random parts have been constructed in a more straightforward way, which performed similar to GSGP but produced smaller and less time-consuming solutions.

The growth problem, which makes solutions not interpretable, and the issue that geometric operators cannot be applied to any problem domain, make this approach not usable for program synthesis.

2.5 Conclusion

This chapter presented an overview of evolutionary algorithms, especially Genetic Programming. Grammars and the usage of grammars in GP have been discussed with a focus on the operators being used in this thesis, like lexicase selection [60] and PTC2 [56]. An introduction to program synthesis was given, and approaches from different fields of research have been briefly described. GP systems that have been applied in the program synthesis space have been discussed as well as their advantages and disadvantages. Afterwards, a detailed introduction of semantics has been given, and multiple semantic operators were presented. Lastly, Geometric Semantic Genetic Programming and some of its issues have been reviewed.

Although grammars have been used in GP to tackle program synthesis, only little research has been undertaken in this domain. Especially, no general concept exists to tackle program synthesis without writing a new grammar for the problem at hand. This shortcoming will be addressed in Chapter 4 by proposing a grammar design approach. To this end, a study on derivation tree based GP will be conducted in Chapter 3. Then the grammar design approach will be further investigated in Chapter 5 and 6. Finally, as already mentioned, seman-

tics has been underutilized in program synthesis as most research in semantics focused on the boolean and regression domain. Chapter 7 examines semantics in program synthesis and explores how to exploit semantic information to improve performance.

Part II

Experimental Research

Chapter 3

General Grammar Design for Derivation Tree Based Genetic Programming Systems

This initial experimental chapter covers grammar design for sorting networks with a focus on derivation tree based Grammar-Guided Genetic Programming (G3P) systems. Grammars created in this chapter can be used by linearised G3P systems as well, but some observations do not hold for all G3P systems and will be pointed out. Although G3P systems and therefore grammars are widely used in the GP research community, the process of designing grammars is an underexplored research area [86]. As a consequence, this chapter aims to give some insights about derivation tree structures constructed by grammars and their influence on search performance to understand grammars that use recursion and produce variable length output. This knowledge is used to create a design pattern for grammars in the program synthesis domain and is discussed in Chapter 4. This chapter is based on work from [118].

3.1 Grammars for G3P

Grammars in G3P systems are used to represent a search space and are often used to limit the search to valid individuals. If required, grammars can also be used to include bias, e.g. expert knowledge, to drive the search into a specific region [17, 119]. The careful design of grammars is therefore of utmost importance, as it

defines the representation and has a strong influence on the search performance of an algorithm. Although some studies on grammars have been undertaken including, but not limited to, Whigham [17], Nicolau [28, 29], Hemberg [26] and Murphy [27], there is still little explicit knowledge of how to design a grammar for a problem. This section sheds some light on grammar design for derivation tree-based genetic programming systems.

For this purpose, a real-world problem, creating sorting networks, is chosen to investigate how different grammars influence the search performance. Grammars for sorting networks are small and only require two to three rules which mainly define the structure of the derivation tree, which is the focus of this chapter, and therefore makes this problem a perfect candidate for this analysis. The study focuses on how different kinds of recursion, which is necessary for variable length output, in grammars affect the search and how GP operators behave on the different grammars.

3.2 Sorting Network

Sorting networks are networks, typically built in hardware, that can sort a fixed number of values in contrast to sorting algorithms in software that allow arbitrarily number of values. A sorting network consists of wires and comparators and it has as many input as output wires. Comparators connect two wires and swap the values if they are not in sorted order. Sorting networks are usually executed on graphics processing units [120].

Figure 3.1 shows an optimal sorting network with four inputs. The wires are drawn as horizontal lines from left, which shows the input, to the right, which shows the output. Comparators are depicted as vertical lines with dots on each end to show which wires are connected. It should be noted that the order of the comparators is essential. For example, putting the rightmost comparator on the left-hand side would not yield a correct output for the input displayed in Figure 3.1.

A sorting network has three properties. The number of inputs, a size and a depth. The number of inputs is given and the size is the number of comparators within the network. The depth of a sorting network is the number of steps it takes to complete the network. At a single step, multiple comparators can be executed at once, as long as there are no comparators that require the same wires as inputs.

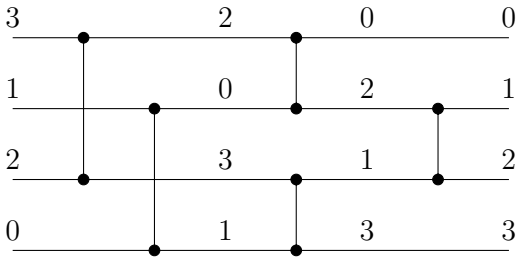


Figure 3.1: Sorting network with four inputs and five comparators. Input values on the left are passed to the right and swapped by comparators if the values are not in sorted order. The output on the right are the input values in sorted order from lowest to highest.

In Figure 3.1, the first two and second two comparators can be executed at once. The last comparator has to be executed on its own as no other comparators are left. Therefore, the sorting network has 4 inputs, 5 comparators and a depth of 3.

Initially, the input values of the sorting network depicted in Figure 3.1 are `[3,1,2,0]`. After the first step, which executes two comparators at once as they do not use the same wires, the values are in the following order `[2,0,3,1]`. The order after the next step which again executes two comparators is `[0,2,1,3]`. Finally, after executing the last comparator the values are in the correct order `[0,1,2,3]`.

Sorting networks have already been tackled with GP [121, 51] and also genetic algorithms [122] before this study. For a range of sorting networks, the optimal size and depth have been proven, or an upper bound has been established [123, 124, 125]. Nevertheless, creating sorting networks is a challenging task.

Testing a sorting network with all possible permutations of inputs to prove its correctness would result in a runtime complexity of $n!$. The runtime complexity can be reduced to 2^n due to the zero-one principle [123], by testing only all combinations of 0, 1. If all combinations of 0, 1 are sorted correctly by a network, all arbitrary values will be sorted correctly as well.

3.3 Structure in Grammars

Variable length representations with grammars require direct or indirect recursion, as shown in section 2.2.8. A direct recursion in a grammar appears when a rule requires its own non-terminal in one of its productions. A simple direct left

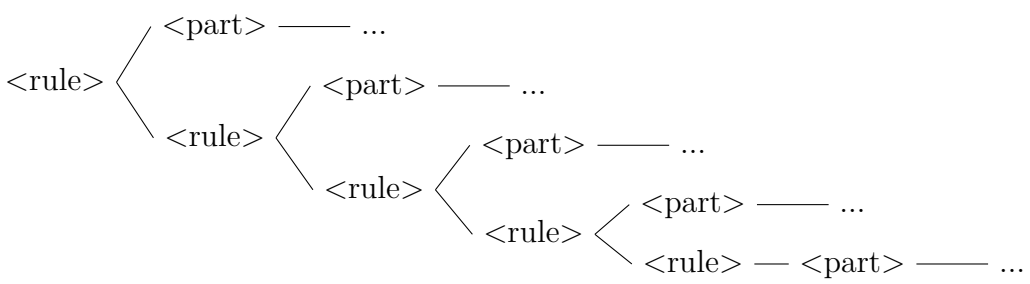


Figure 3.2: Possible derivation tree of a direct left recursion (`<rule> ::= <rule><part> | <part>`). Three dots depicting further subtrees.

recursion, e.g. `<rule> ::= <rule><part> | <part>`, can create variable length representations. `<rule>` and `<part>` can be replaced with any non-terminal symbol. `<part>` could even be replaced with a terminal symbol. Many researchers use direct left recursion, because it is intuitive, but do not argue if it is the best representation. Some examples using this left recursion with other non-terminals symbols include `<code>` and `<line>` for Santa Fe Ant Trail problem [18], `<for>` and `<code>` for program synthesis [13], `<design>` and `<component>` for creating designs [126], `<int_constant>` and `<number>` for integer constant creation [127].

When generating a sentence with such a grammar, the derivation tree for that sentence is more similar to a list than a tree, which is depicted in Figure 3.2. `<rule>` will always produce another `<rule>` non-terminal symbol until the recursion stops, which produces a list of `<part>` non-terminal symbols that contain subtrees with terminal symbols. When operating on derivation trees in G3P, in most cases the same or similar operators as in standard GP are used. As these operators were designed to operate on trees, in many cases n-ary trees, the question is, if such a list-like derivation tree structure is well suited to result in good search performance.

For example, standard crossover cannot exchange two `<part>` nodes, unless they are the last two, between two parents. It only has the possibility to exchange single `<part>` nodes and their respective subtrees or, when operating on a `<rule>` node, crossover can only decide up to which `<rule>` node the derivation tree of the first parent is kept, while the rest gets replaced with what is selected from the second parent. Crossover on a `<rule>` node seems similar to single-point crossover in a GA due to the list-like derivation tree structure. Standard crossover cannot exchange multiple `<part>` nodes from within the “list”. Standard subtree mutation

faces the same issue. When selecting a `<rule>` node, everything afterwards gets replaced or if a `<part>` node is selected then only a single `<part>` node is replaced. It is not possible to change two or more `<part>` nodes from within the derivation tree.

To this end, grammars and their corresponding derivation tree structures are analysed in this chapter to investigate how GP operators behave on different derivation tree structures and if different structures influence the search performance. As subtree crossover and subtree mutation cannot exchange multiple non-recursive parts within a derivation tree on list-like structures, while this is possible with tree-like structures, it is assumed that derivation trees in the shape of trees are better suited as a representation and may improve performance.

3.4 Sorting Network Grammar Design

Sorting networks can be represented as a list of integer pairs. Every pair of integers represents one comparator. The integers of such a pair are the indices of the wires the comparator connects. Therefore, a grammar for this problem must be able to represent every possible number of integer pairs and every possible combination of two integers, where the integers are limited to the indices of the number of wires of a given sorting network.

Table 3.1 show five different grammars for a sorting network with four inputs. The grammars can produce the same sentences, although the derivation trees generated as an intermediate process will be different in structure. The number of inputs for the sorting network can be easily adapted by adding additional indices to the rule `<node>`. The grammars have simply been named from G1 to G5.

G1: The first grammar G1 contains a start symbol `snet`, for sorting network, and a simple direct left recursion, as explained in section 3.3. The non-terminal `<part>` has been replaced with two non-terminals `<node>`. Therefore, every recursion of `<snet>` adds a pair of integers or one comparator in terms of the sorting network. A derivation tree for the optimal sorting network with four inputs, shown in Figure 3.1, is depicted in Figure 3.3. As G1 only uses a left recursion, the derivation tree can only expand on the left-hand side, which leads to a similar list-like structure as with the previously seen derivation tree in section 3.3. As already explained, crossover and mutation are not

Table 3.1: Five different grammars for sorting networks with four inputs, which all generate the same language.

Name	BNF
G1:	`<snet> ::= <snet> <node> <node> | <node> <node>` `<node> ::= 0 | 1 | 2 | 3`

G2:	`<snet> ::= <snet> <nodes> | <nodes>`
	`<nodes> ::= <node> <node>`
	`<node> ::= 0 | 1 | 2 | 3`

G3:	`<snet> ::= <snet> <node> <node> | <node> <node> <snet>`
	`| <snet> <node> <node> <snet> | <node> <node>`
	`<node> ::= 0 | 1 | 2 | 3`

G4:	`<snet> ::= <snet> <nodes> | <nodes> <snet>`
	`| <snet> <nodes> <snet> | <nodes>`
	`<nodes> ::= <node> <node>`
	`<node> ::= 0 | 1 | 2 | 3`

G5:	`<snet> ::= <snet> <snet> | <node> <node>` `<node> ::= 0 | 1 | 2 | 3`

able to change multiple pairs of integers within the tree at once due to this list-like structure, unless they are the last ones to the left-hand side of the derivation tree.

G2: Grammar G2 is similar to G1 with the only addition of an extra rule `<nodes>`, which encapsulates a pair of integers. Therefore, G2 is not only more similar to the previously discussed grammar rule `<rule> ::= <rule><part> | <part>`, but the additional rule `<nodes>` allows operators to exchange a pair of integers instead of single integer values in G1.

G3: Grammar G3 is the first grammar that creates a more tree-like structure. Additional to the left recursion in G1, a right recursion was added. This change alone would still create a list-like structure, where the derivation tree could either expand left or right. For the tree-like structure, the production rule `<snet> <node> <node> <snet>` is required, as it allows expansions of the derivation tree in two directions. Every production of `<snet>` adds a pair of nodes as before.

A derivation tree created with G3 for the optimal four input sorting network is depicted in Figure 3.4. Whereas with G1 there was only a single derivation

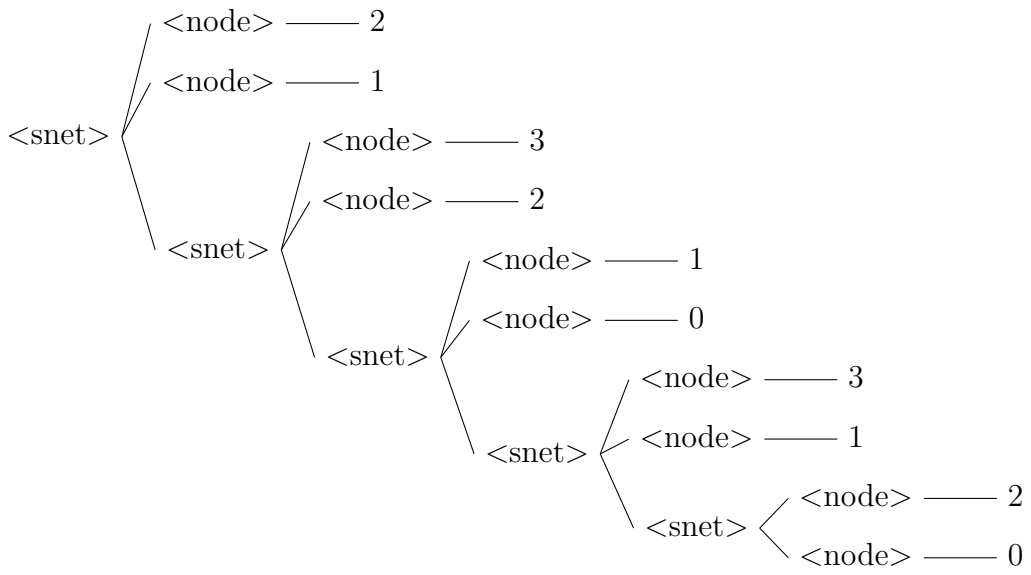


Figure 3.3: G1 derivation tree for the optimal sorting network with four inputs shown in Figure 3.1.

tree that can express the optimal sorting network with 4 inputs, with G3 multiple derivation trees, exist that can express the same sorting network.

G4: Similar to G2, a grammar G4 was created from G3 that encapsulates a pair of integers in an additional rule `<nodes>` with the same benefit as before that a pair of nodes can be exchanged at once. Even with the tree-like structure that makes a difference, because when looking at Figure 3.4 a single pair of integers can only be exchanged in G3 when `<snet>` derives

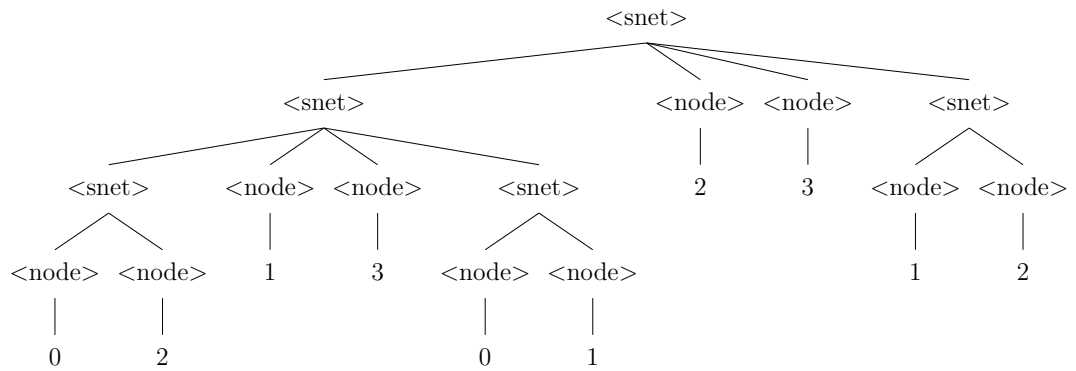


Figure 3.4: G3 derivation tree for the optimal sorting network with four inputs shown in Figure 3.1.

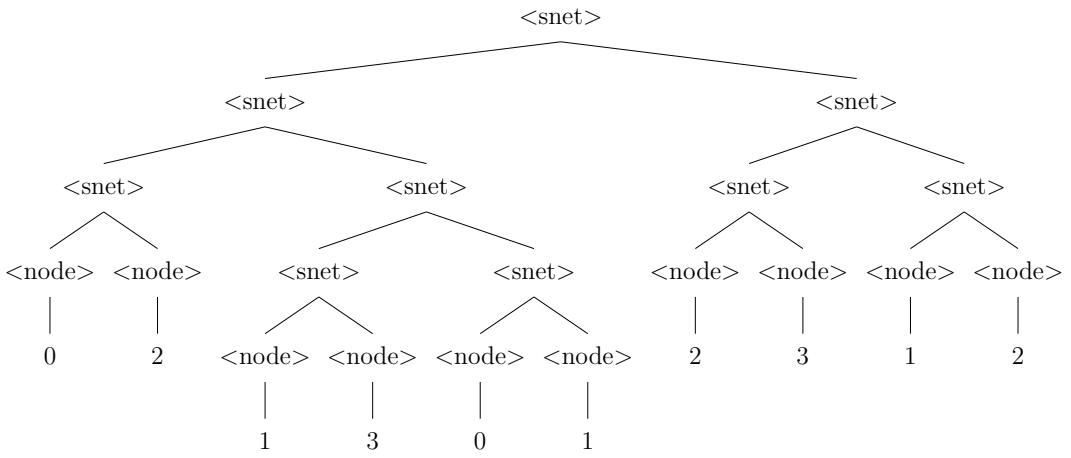


Figure 3.5: G5 derivation tree for the optimal sorting network with four inputs shown in Figure 3.1.

into `<node><node>`. Otherwise, multiple pairs will be exchanged at once. For both, G3 and G4, GP operators can manipulate multiple pairs of nodes within the tree, even if they are not the first or last pair. In terms of the sorting network, multiple comparators can be changed at once that are not executed at the first or last step. Therefore, crossover and mutation might not be as destructive as in G1 or G2 were everything after a selected `<snet>` node in the derivation tree will be changed.

G5: At last, grammar G5 might not be the most intuitive choice as a variable length representation, but as G3 and G4 it will create tree-like structure, which is depicted in Figure 3.5. The derivation tree created is a binary tree. G5 could be adapted to create n-ary trees by adding additional `<snet>` non-terminals to the first production of `<snet>`. In contrast to G1 and G3, there is no need to create a grammar with the non-terminal symbol `<nodes>` to encapsulate a pair of integers, as single pairs of integers can already be exchanged by choosing a `<snet>` node that derives into two `<node>` nodes. But similar to G3 and G4, GP operators can manipulate any number of pairs of integers within the derivation tree due to the tree-like structure. Therefore, G5 seems to combine the benefits of tree-like structures from G3 and G4 and the possibility to exchange single comparators in one operation from G2 and G4.

Table 3.2: Minimum number of nodes and minimum depth for each grammar given a certain number of comparisons (c)

Grammar	Number of nodes	Depth
G1	$3 * c$	$c + 1$
G2	$4 * c$	$c + 2$
G3	$3 * c$	$\lceil \log_2(c + 1) \rceil + 1$
G4	$4 * c$	$\lceil \log_2(c + 1) \rceil + 2$
G5	$4 * c - 1$	$\lceil \log_2(c) \rceil + 1$

3.4.1 Derivation tree sizes

Derivation tree-based G3P systems have to limit the tree sizes either by depth or number of nodes as otherwise, trees could theoretically grow infinitely big, as in standard GP. As the grammars G1-G5 create different derivation tree structures, representing the same sorting network with G1-G5 requires a different number of nodes and depths, as can be seen in Figure 3.3, 3.4, 3.5. The sorting networks are represented by the grammars as comparators that are executed. The number of nodes and the minimum depth for a derivation tree per grammar to represent a certain number of comparators can be calculated per grammar and is shown in Table 3.2. Note that in contrast to the derivation tree representation used in the figures, the G3P system used for the experiments represents a production as a single node with as many subtrees as non-terminals are contained in the subtree. Representing a production as a single node reduces the number of nodes required within a tree for the G3P system but does not change the behaviour of the derivation trees or search operators.

As can be seen in Table 3.2, grammars that create a list-like structure grow linearly in depth, while grammars that produce a tree-like structure grow only logarithmically. There is little difference in the number of nodes required per comparator added to a derivation tree. G1 and G3 require fewer nodes to represent the same number of comparators as the other grammars, because they do not have the additional rule `<nodes>`, which also implicitly exists in G5. These formulas will be used for the experiments to allow all grammars to be able to create derivation trees with the same number of comparators.

```
<nodes> ::= 0 1 | 0 2 | 0 3
          | 1 2 | 1 3
          | 2 3
```

Figure 3.6: All possible comparisons in a sorting network with four inputs.

3.4.2 Grammar Design Details

For linearised G3P systems, other properties of the grammars might be important as well. For example, G3 and G4 have three productions in the rule `<snet>` that add another recursion and only one production that stops the recursion. So there is only a 25% chance that recursion will stop, while 75% of the time the recursion will continue or even add a second recursion. Mapping individuals as it is done in GE, might often lead to invalid individuals. As a derivation tree based G3P system will be used, which stops recursion due to a tree depth or tree node limit, if the tree would grow further than those limits, these problems are less of a concern.

Another detail about the grammars that should be noted is that the rule `<nodes>` is a so-called unit production as it has only a single production that it can be expanded to. Depending on the G3P system used, especially linearised G3P systems, the rule might automatically be expanded, or all `<nodes>` non-terminals will be replaced with its production. This will lead to grammars G1 and G2 as well as G3 and G4 being treated identically and will not change anything in the search performance.

Further improvements could be made to the grammar by reducing the combinations of integer pairs that can be created. The rule `<nodes>` can create any combination of integers given by the rule `<node>`, which results in n^2 combinations. The problem is that many of these combinations are superfluous. 3 1 and 1 3 represent the same comparator and 1 1 would be a comparator with the same input twice. The rule `<nodes>` could be changed to only contain valid combinations and remove duplicate comparators, so that number of combinations is reduced to $\frac{n^2-n}{2}$. Figure 3.6 shows the rule `<nodes>` with the reduced number of combinations for 4 inputs. This change would reduce the search space, but would also remove the possibility of removing a comparator by changing its input to the same wire, as those will be removed before executing the sorting network.

The reason why this possible optimisation has not been investigated is that it is a problem specific adaption, while all other changes to grammars that have

been made, like encapsulating non-terminals in an additional rule or using tree-like derivation trees rather than list-like, can be applied to other grammars as well.

3.5 Experimental Setup

The experiments are in two parts to answer two questions. First, how do the different grammars influence the amount of genetic material changed by the search operators? Especially, do the grammars that produce tree-like structures help the GP operators exchange less genetic material in one operation as is the assumption form section 3.4. This is of interest, because it is assumed that subtree operators will achieve worse performance on derivation trees with a list-like structures due to the inability of exchanging subtrees with multiple non-recursive parts within the tree. Therefore, the amount of genetic material exchanged on different grammars might give an indication over performance. The second question is, do the different grammars, again especially the tree-like structures, influence the search performance and can conclusions be drawn about general grammar design, especially recursion in grammars?

3.5.1 Experiment 1

To answer the first question, the first experiment does not use any fitness evaluation and only checks the amount of genetic material, measured in the number of nodes of a derivation tree, which gets exchanged by the GP operators. To this end, each grammar is used three times. One time the experiment is run only with crossover, a second time just with mutation and a third time with both search operators. 100 runs are executed per grammar and per search operator combination. The probability of using each operator is set to 100% each time. As no fitness evaluation takes place, random selection is applied. The sorting networks created are limited to 50 comparators. This limit is implemented by restricting the number of nodes in a tree per grammar as calculated by Table 3.2.

3.5.2 Experiment 2

In the second experiment, fitness evaluations are used to identify differences in search performance in the grammars. A sorting network with 12 inputs is chosen to

Table 3.3: Experimental parameter settings. [1]Experiment 1. [2]Experiment 2.

Parameter	Setting
Runs	100^1, 50^2
Generations	100
Population size	1000
Population initialisation	PTC2 [56]
Tournament size	7
Internal Crossover probability	0.9
Mutation probability	$100\%^1$, $5\%^2$
Elite size	0^1, 1^2
Maximum compare-exchange operations	50^1, 59^2

compare the grammars. Therefore, 4096 fitness cases have to be tested to establish the correctness of the network. The same fitness function as in Koza et al. [121] is used, which minimises the number of incorrect fitness cases plus the number of used comparators divided by 100. So, the primary objective is to correctly sort as many fitness cases correctly as possible, while the secondary goal is to use as few as possible comparators. The optimal sorting network requires between 37 and 39 comparators as has been proven in [124]. The number of comparators for this experiment has been limited to 59, which is the proven upper bound required times 1.5 rounded up.

3.5.3 General Settings

The parameter settings for both experiments are summarised in Table 3.3, with differences marked with superscripts. The elite size in the first experiment is zero as no evaluation is used.

In both experiments, the number of nodes in a derivation has been restricted as it allows more fine-grained control of the number of comparators that can be created with one individual in contrast to the tree depth, which may enable grammars to produce a tree-like structure to use more comparators than the others. Also, Daida et al. [55] showed that GP tends to evolve rather sparse binary trees than dense ones, which also encourages to restrict the number of nodes in a tree rather than the tree depth. Therefore, Probabilistic Tree-Creation 2 (PTC2) [56] has been used, as in all experiments in this thesis, to initialize the first generation, as PTC2 uniformly samples from trees of different length and the

maximum number of nodes can be set, while other initialization strategies like, e.g. ramped half-and-half [49] are only able to restrict the tree depth.

For these experiments, it is also important to note that the in the GP community widely adopted Koza style crossover [49] is used. This type of crossover does not choose uniformly from all possible nodes in a tree but selects 90% of the time an internal node and only 10% of the time a leaf node. The reason is that trees used in GP usually have a branching factor, the number of subtrees per node, of two or higher. Therefore, a tree will have more leaf nodes than internal nodes and the chance of selecting a leaf node with crossover is high, which leads to crossover only exchanging very small subtrees. Koza style crossover counteracts that behaviour by having a high probability of selecting an internal node. This is especially of interest for the second experiment and will be discussed in section 3.6.2.

3.6 Results

In this section, the results of the sorting network experiments with grammars that produce a different kind of structures will be discussed.

3.6.1 Experiment 1

The general idea of using different grammars for variable length representations which produce the same language emerged from the problem that standard GP operators are designed for trees and with list-like structures as produced by G1 and G2 too much genetic material might get exchanged at once. The first set of experiments confirms that hypothesis. Figure 3.7 shows the number of nodes that get added and removed with a single crossover and mutation operation on average over generations. Figure 3.7 only shows the results when using crossover and mutation at the same time. The results gained when only using crossover or mutation are very similar and do not add any further insight, hence they have been omitted. The only difference when using crossover without mutation is that trees shrink over the first 30 generations and then stabilises. The reason is that while the node selected for crossover in the first parent is selected entirely at random, the subtree selected in the second parent has to obey the number of node restrictions allowed in a single tree. Therefore, the subtree selected in the second

parent has a higher chance of being smaller. While this phenomenon stabilises after 30 generations when using crossover on its own, it can only be observed up to generation 10 when using crossover and mutation. Mutation counteracts crossover by adding slightly bigger subtrees than it removes, probably because it is applied after crossover. When using mutation on its own, a similar amount of genetic material is removed and added.

Figure 3.7 confirms that genetic operators add and remove bigger subtrees when grammars are used that produce a list-like structure, like G1 and G2. When a node is changed that defines the structure, in these experiments `<snet>`, then all the genetic material after that node will be changed. In case of grammars that produce a list-like structure, every level of depth in the tree only decreases the number of nodes linearly. Therefore, there are more possible nodes that have large subtrees, while every level of depth with grammars that produce tree-like structures can reduce the number of nodes by half. Derivation trees created with grammars that produce tree-like structures contain more nodes with smaller subtrees compared to the list-like structures with the same number of comparators. Crossover and mutation add and remove less genetic material in case of G3, G4 and G5 as expected.

In case of crossover, a difference of genetic material added and removed between G1 and G2 can be seen that is not apparent with mutation. This appears in crossover because G2 has the extra rule `<nodes>`. Therefore, G2 has more internal nodes and `<nodes>` contains smaller subtrees compared to `<snet>`. Mutation selects randomly from all nodes in a derivation tree, while crossover favours internal nodes, as explained in section 3.5.

3.6.2 Experiment 2

For the second experiment fitness evaluation was turned on to analyse the performance of grammars that produce different structures. Table 3.4, 3.5 and 3.6 show the results on different sorting networks or using different operators. All tables show a p-value calculated using Wilcoxon rank sum test of the best fitness over 50 runs between two grammars. A significance level $\alpha = 0.05$ is used with Bonferroni correction of $m = 10$, which results in $\alpha = 0.05/10 = 0.005$. Statistically significant differences are marked in bold. Note that the matrix of p-values is symmetrical and that upper triangular matrix has been removed for readability.

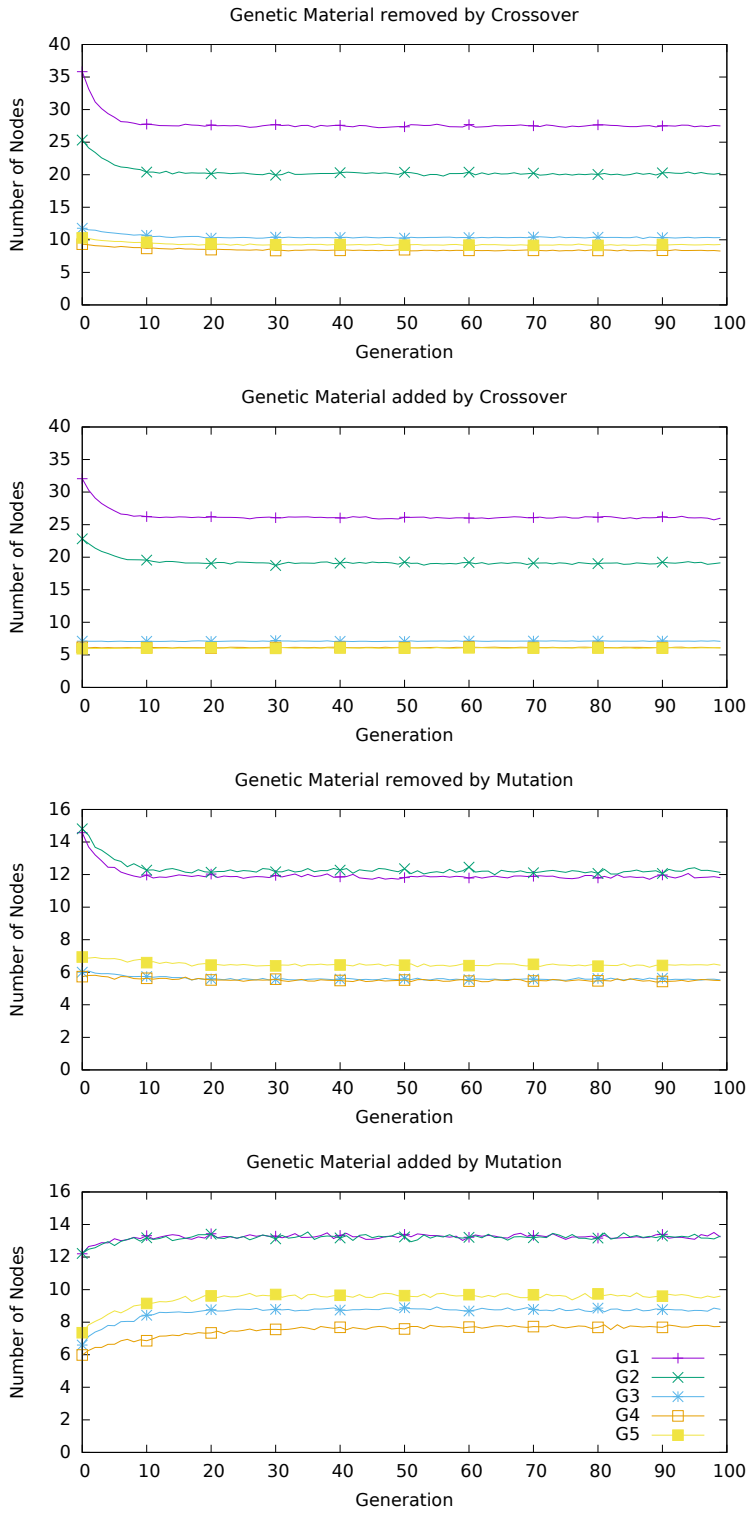


Figure 3.7: Genetic material added and removed when using crossover and mutation. Note that the y-axis scales for crossover and mutation are different as mutation tends to exchange smaller amounts of genetic material.

Table 3.4: Results for sorting networks with 12 inputs with the average best fitness, standard deviation, best individual and success ratio over 50 runs as well as the p-value of a Wilcoxon rank sum test of the best fitness between two grammars.

	Avg fit. ± Std dev	Best	Succ.	G1	G2	G3	G4	G5
G1	82.60 ± 42.79	18.56	0%	–				
G2	31.85 ± 21.66	0.58	4%	**0.000**	–			
G3	65.60 ± 47.89	14.58	0%	0.015	**0.000**	–		
G4	23.53 ± 15.47	0.53	8%	**0.000**	0.069	**0.000**	–	
G5	34.13 ± 28.26	0.55	2%	**0.000**	0.962	**0.000**	0.075	–

Table 3.4 summarizes the results for sorting networks with 12 inputs. Only three of the five grammars are able to evolve sorting networks that successfully sort all inputs correctly, even though substantially more comparators are used as an optimal network would need. Interestingly the best results are achieved by G2, G4 and G5, where G2 generates list-like and G4 and G5 tree-like structures. G2 and G4 contain the extra rule `<nodes>`. So for a fair comparison between list-like and tree-like structures G1 and G3 as well as G2 and G4 have to be compared. In both cases, the tree-like do slightly better but none shows a statistically significant difference.

The experiment has been repeated with a more complex instance of the problem, with a sorting network with 14 inputs which increases the number of training cases to 16,384. Also, the maximum number of comparators allowed has been increased, depending on the grammar used and Table 3.2. The results are summarized in Table 3.5. No successful sorting network was found for 14 input with any of the grammars. The overall results are similar to the experiment with sorting networks with 12 inputs, even the p-values behave similarly. G1 and G3 do worse than the other grammars. The tree-like structure of G3-G5 seems not to be of an advantage. Otherwise, G3 would perform better and G2 worse.

The grammar G2 and G4 contain the extra rule `<nodes>`, which implicitly exists in G5 as explained in section 3.4. So the overall structure of the derivation tree has little effect while having the possibility of exchanging complete comparators with a single operation of crossover or mutation gives the grammars G2, G4 and G5 a major advantage. As the Koza style crossover was used in the experiments that favours selecting internal nodes for crossover, this might explain why these grammars performed better. To prove this hypothesis, a last experiment is executed where a subtree crossover is used that uniformly selects from all nodes at

Table 3.5: Results for sorting networks with 14 inputs with the average best fitness, standard deviation and best individual over 50 runs as well as the p-value of a Wilcoxon rank sum test of the best fitness between two grammars.

	Avg fit. ± Std dev	Best	G1	G2	G3	G4	G5
G1	769.83 ± 303.51	256.74	–				
G2	245.52 ± 143.94	26.77	**0.000**	–			
G3	575.55 ± 300.54	102.75	**0.002**	**0.000**	–		
G4	230.20 ± 119.65	8.73	**0.000**	0.736	**0.000**	–	
G5	297.68 ± 197.86	40.75	**0.000**	0.282	**0.000**	0.169	–

Table 3.6: Results for sorting network with 12 inputs with subtree crossover that chooses from all nodes in the tree with equal probability. The Table shows the average best fitness, standard deviation and best individual over 50 runs as well as the p-value of a Wilcoxon rank sum test of the best fitness between two grammars.

	Avg fit. ± Std dev	Best	Succ.	G1	G2	G3	G4	G5
G1	9.72 ± 8.62	0.50	20 %	–				
G2	11.81 ± 11.21	0.53	22 %	0.471	–			
G3	11.77 ± 12.83	0.53	18 %	0.863	0.815	–		
G4	12.73 ± 11.41	0.53	16 %	0.198	0.694	0.361	–	
G5	13.29 ± 11.91	0.52	16 %	0.123	0.517	0.370	0.836	–

random. The experiment is executed on the sorting network with 12 inputs and the results are shown in Table 3.6. The percentage of runs that found a network that successfully sorts all possible inputs drastically increased for all grammars and no statistically significant difference between any grammars can be found in contrast to the previous experiments. All grammars achieve similar performance by using a crossover that does not favour internal nodes in the tree. Due to this change, crossover is likely using leaf nodes more often and therefore it only exchanges single nodes. The structure of the derivation tree does not matter as much anymore.

Allowing crossover to happen more often on the leave nodes appears to be a benefit for the sorting network problem, but no conclusion can be drawn for other problems. In case of many other problems which use recursion, see section 3.3, the non-recursive part creates bigger subtrees than just single node with a number. In such a case using crossover as in the last experiment have a negative effect.

In terms of the structure of the grammar, G2, G4 and G5 had a positive effect as can be seen in Table 3.4 and Table 3.5, because the non-recursive part, a single comparator, could be exchanged in a single operation, which G1 and G3 lacked.

3.7 Summary

In this chapter, grammars that produce different structures in derivation trees have been investigated. Grammars have been classified into those that generate list-like derivation trees and tree-like. Five different grammars which produce the same language have been created in total. All of which produced a variable number of comparators for sorting networks. The grammars have been analysed in terms of how genetic operators, crossover and mutation, behave, especially how much genetic material gets exchanged in a single operation. Crossover and mutation exchanged less genetic material in one operation on average if the derivation tree was created with a grammar that produces tree-like structures, as was expected.

A second set of experiments was conducted to investigate performance differences in terms of the structure grammars produced. It was found that list-like and tree-like structures performed equally well. Grammars that offered the ability to exchange the non-recursive part in the recursive rule with one operation outperformed the grammars which did not provide that possibility. Another observation was that Koza-style crossover had an adverse effect on performance in the case of sorting networks as it favours internal nodes, but this behaviour is likely to be problem specific, and no overall conclusion can be inferred.

The experiments showed that even though the structures that are produced by grammars affect the amount of genetic material that gets exchanged by GP operators, it has little or no effect on the overall performance with derivation tree-based G3P systems. This suggests that no particular attention has to be paid on the structure when using recursive rules in grammars. These observations will most likely not hold true for linearised G3P systems, and separate experiments would need to be conducted. Also, further investigations might be required to analyse if grammars with non-recursive parts that create bigger subtrees than the rule `<nodes>` exhibit similar behaviour.

In the next chapter, a grammar design pattern for tackling arbitrary program synthesis problems is developed and how derivation tree structures influence search

performance with grammars for program synthesis is further investigated on a general program synthesis benchmark suite.

Chapter 4

A Grammar Design Pattern for Arbitrary Program Synthesis Problems

In this chapter, a grammar design pattern for program synthesis is discussed. The grammar design pattern offers the ability to create grammars in arbitrary languages that are reusable for arbitrary program synthesis problems. Grammars have been used to tackle problems from the program synthesis domain, but usually, the designed grammars were tailored to a specific problem resulting in bespoke grammars, which cannot be reused. The shortcomings of other systems, especially GP systems, are examined and addressed with a flexible grammar-based approach. The grammar design approach is tested on the general program synthesis benchmark suite, presented in Section 2.3.2, and compared to PushGP as it is the state of the art system that has been used to tackle the problems from the benchmark suite. Insights from the previous chapter are used while designing the grammars and further experiments on the structures grammars create are conducted. This chapter is based on research first published in [128].

4.1 Previous Approaches to Program Synthesis

This section recaps previous approaches to program synthesis in GP to highlight shortcomings and promote a new grammar design approach. Advantages and disadvantages of each approach are discussed. More detail on these and other

approaches can be found in Section 2.3.1. The main approaches to consider for program synthesis in GP that create code from scratch are Grammar-Guided Genetic Programming (G3P), Strongly Formed Genetic Programming (SFGP) and PushGP. G3P will also be used by the grammar design approach introduced in this chapter. Strongly Formed Genetic Programming applies restriction on nodes to provide type safety and has similarities to the grammar design approach. PushGP is the state of the art for program synthesis in GP and has been the first system that has been tested on the general program synthesis benchmark suite.

4.1.1 Grammar-Guided Genetic Programming

G3P systems, like Grammatical Evolution [18] or CFG-GP [17], have been used to evolve code for program synthesis in arbitrary languages. Grammars are very flexible and can be used to represent the search space of a wide range of problems, evolving music [20], creating truss design [21], optimising pylon structures [22], evolving aircraft models [23], controlling femtocell network coverage [24] and program synthesis [13]. A G3P system does not require any additional code to evolve a solution, except a grammar and a fitness function that can evaluate individuals created by the grammar. The code that is evolved in G3P can be in an arbitrary programming language, which is defined by the grammar.

Even though G3P systems are perfect candidates for tackling problems in the program synthesis domain, so far they have been underutilised. No general concept for grammars exists that can tackle arbitrary problems. Every grammar that has been used to evolve programs has been tailored to a single specific problem and cannot be reused without changes being made. In Section 4.2 a grammar design pattern will be introduced to solve these shortcomings. The grammar design pattern will provide a way to create reusable grammars for many program languages as well as an easy way to provide fitness evaluation to G3P for arbitrary program synthesis problems.

4.1.2 Strongly Formed Genetic Programming

Strongly Formed Genetic Programming (SFGP) [96] is an extension of Strongly Typed Genetic Programming (STGP) [12]. While standard GP operates on the premise that a function in a node must be able to handle any input from its subtrees, STGP changed the behaviour of its genetic operators (initialisation,

crossover and mutation) to adhere the types, specifically node-types and data-types, available in STGP. Data-types are similar to data-types in programming languages, like integers or strings. Node-types, such as `Statement`, `CodeBlock` or `Loop`, are used to restrict the shape and structure of a program. Every function in STGP outputs a specific type and requires specific types as input. Its genetic operators are not allowed to create or change any (sub-)trees that are not coherent with the required types. Therefore, evolved individuals are type consistent, which is important for program synthesis as many different data types may be used.

SFGP extends STGP by adding additional nodes that allow the generation of generic code, like nodes for control flow (e.g.`CodeBlock` or `ForLoop`) and variables (`Variable`). Every node needs to be implemented within the SFGP system. Many aspects in SFGP have to be predefined, like a fixed number of statements/subtrees per code block, so they do not have a variable length and the number of variables as they are part of the terminals. SFGP only evolves programs in its own structure, which can be viewed as pseudocode, which needs to be translated to be used in a real-world program.

Although SFGP is an extension to STGP that allows evolving code for program synthesis problems, it exhibits many limitations. The evolved code cannot be used in a real-world program. Any new functionality has to be implemented in the SFGP system and cannot be transferred to any other system, unlike grammars, that can be used by any G3P system.

4.1.3 PushGP

PushGP [95] evolves programs in the Push programming language that was solely developed for research in the field of evolutionary computation. Implementations of Push and PushGP are available in many programming languages including Python and C#. The reference implementation is in Clojure.

In contrast to many common programming languages, Push uses a stack-based execution architecture. It has a stack per data type and additionally one for instructions that are going to be executed. Due to this architecture, Push does not need variables as data is stored on the corresponding stack and the execution of code can be manipulated at runtime as the stack containing instructions can be changed as well.

PushGP offers many advantages over the previously discussed systems. There is no need to declare the number of variables beforehand, as data is stored on stacks. All available instructions to PushGP are implemented in the corresponding programming language in a protected way which avoids runtime exceptions. By using wrapper functions, functionality and libraries of the programming language, PushGP is run on, can be made available during evolution.

As Push is a language not used outside the research community, Push code cannot directly be integrated into existing software. The Push interpreter is required as well, which adds overhead. Push code can be run in different programming languages, given an interpreter, but only if the core Push functionality is used. This means wrapper functions to use libraries from the programming language PushGP was run on, cannot be executed on any other Push interpreter.

4.1.4 Summary

All of the systems mentioned above are capable of tackling general program synthesis problems. But all of them come with certain limitations. PushGP is the most flexible of the systems as grammars have to be tailored at the moment and Push provides an extensive built-in set of functionality, although Push is not used outside the research community and an interpreter is required to run the code. An approach to evolve code in a target programming language without having to adapt the function set per problem by hand and without reimplementing functions or wrapping them would provide a flexible way to approach program synthesis in research and industry.

4.2 System Description

The concept proposed to address shortcomings identified in the previous section is using a grammar design approach. The choice to use grammars was made because no modification is required to use a grammar in a G3P system, while to add functionality in other systems code has to be changed within the system. So, grammars can be exchanged between systems and researchers without the need of changing one's own system, given that the same format is used, Backus–Naur form (BNF) in many systems [128, 39, 129]. Additionally, grammars can be

easily extended if they are well structured, even with little or no knowledge of programming, which makes them easy to use for newcomers as well.

The grammar design approach aims to generate code for arbitrary program synthesis problems, to be flexible and extensible to add more functionality when required, to allow using libraries of the target programming language and having the possibility to apply this approach to a number of program languages. Additionally, it should be relatively simple to provide fitness evaluation to a G3P system for the program synthesis problems. Otherwise, the simplicity gained with the grammars would be wasted.

The approach presented in this thesis consists of two parts that have to be provided to tackle program synthesis problems. The first part is the grammar design pattern that outlines how to design grammars to tackle arbitrary program synthesis problems, see Section 4.2.1, and the second part is a skeleton that provides protected methods and a fitness function, which is explained in detail in Section 4.2.2. An important property in GP is closure [49], which consists of type consistency and evaluation safety. Closure is required for GP to work effectively. The grammar guarantees type consistency, whereas the skeleton provides evaluation safety.

The code that is going to be evolved with the grammars created in this thesis is not abstract code or pseudocode, but specific for a target programming language. As there exist syntactical differences between programming languages, grammars and skeletons need to be adapted for other languages. For a single language, the grammars have to be created only once. Advantages and disadvantages compared to other systems will be pointed out in Section 4.2.3. In this thesis, all the code that is going to be evolved will be in Python, but the general concept of grammar design applies to a variety of programming languages. Specific differences made for Python will be outlined in Section 4.2.4. Due to exceptions during evaluations, programs may not be evaluated, as can happen in other systems as well, and will be discussed in Section 4.2.5.

4.2.1 Grammar Design Pattern

The first part of the grammar design approach is the grammar design pattern. Grammars are used to specify all possible solutions that can be created and therefore define the search space. The design pattern presented here enables the cre-

ation of grammars that can tackle arbitrary program synthesis problems. While grammars for all programming languages exist to be able to parse them for compilation or interpretation, these grammars are not suited for evolving programs. These grammars contain the full language specification, which will contain substantially more functionality than is usually used to tackle program synthesis and will make the search space infeasible. Additionally, these grammars do not contain function or variable names, which are required by most GP systems to be defined beforehand. Even when providing variable names, these grammars would not prohibit using variables before declaring which leads to error during execution.

Due to these issues, the idea is to create not a single grammar for every possible problem, but a range of grammars that can be combined depending on the requirements of the problem. In contrary to other problem domains, like, e.g. regression that only uses floating point values, program synthesis problems usually require multiple different data types. Although, regression can be seen as a subproblem of program synthesis. Using all possible data types to solve a problem that only requires a few increases the search space significantly. So, the idea is to produce a number of grammars that can be combined to fit the requirements of the problem at hand. One grammar per data type will be created as well as one general grammar which contains the structure of the program.

Figure 4.1 displays the grammars that have to be created and Figure 4.2 shows an example of the bool.bnf created for the boolean data type in Python. Note that multiple grammars for lists are necessary, although lists could contain elements of different types in many programming languages. The reasons are that these grammars guarantee type safety to achieve closure. All elements in a list of list_integer.bnf are of type integer. Grammars are suited to address type consistency, which is important because "type awareness reduces the search space and makes genetic operators more effective" [100]. Not every function in a programming language is defined for each data type. Applying an append operation on an integer would result in an exception, either during runtime if the language is interpreted or the program would not even compile. Through proper design of grammars, these problems can be avoided, and it can be assured that all individuals evolved with the grammars are going to be syntactically correct. For this purpose, each grammar contains a rule that returns a value, like bool.bnf `<bool>`, which defines all possible operations that operate with boolean values. Another rule for variable definition is required. In case of bool.bnf, it is `<bool_var>`, which

structure.bnf	
bool.bnf	list_bool.bnf
integer.bnf	list_integer.bnf
float.bnf	list_float.bnf
string.bnf	list_string.bnf

Figure 4.1: Grammars per data type.

```
<bool_assign> ::= <bool_var>' = '<bool>
<bool_var> ::= 'b0' | 'b1' | 'b2'
<bool> ::= <bool_var> | <bool_const>
         | <bool_pre> <bool>
         | '(' <bool> <bool_op> <bool> ')'
<bool_const> ::= 'True' | 'False'
<bool_pre> ::= 'not'
<bool_op> ::= 'and' | 'or'
```

Figure 4.2: Boolean grammar (bool.bnf)

contains all boolean variables that can be used. The last rule that is required is one to assign values to variables, `<bool_assign>` in case of bool.bnf as shown in Figure 4.2. All grammars are defined in the context-free Backus-–Naur Form (BNF).

Functions might require multiple different data types as input and output. For example, the comparison of two integers results in a boolean value. Such operations can be added to either grammar. Before using the grammars, they have to be combined depending on the data types required for a problem. This process can easily be automated, and additional steps can be taken. One is to remove functions for which not all possible data types are available. For example, the integer comparison operation can be removed if the boolean data type is not used. Another step is to remove unit-productions, rules that only contain a single production. Every instance of the non-terminal of that rule can be replaced with its production as there is only one option. During the process of combining grammars, the number of variables available and input and output variables have to be added as well. The data types of the input and output of a program are known beforehand, and variables for those can be added to the corresponding grammars. The number of variables required for a program to store temporary data is most likely not known, which poses a problem, as individuals can only consist of what is defined in the grammars. Therefore the number of variables has

to be defined beforehand unless the grammar is changed during the evolutionary process [130]. In the example in Figure 4.2, `<bool_var>` has three variables, but more could be added.

The grammars created in this thesis are aimed to tackle general program synthesis problems without being tailored to any specific problem. Nevertheless, grammars provide one with the possibility to easily adapt them if required for a particular case. This could either be by adding bias to use a specific function, by adding it more often to the grammar and therefore increasing the chance of it being used or by adding complete code snippets. Another advantage of grammars is that functionality of any library available in a programming language can be added if necessary to evolve even more complex programs, instead of evolving everything from scratch.

As the grammar design pattern introduced in this section, is targeted to evolving code for a single programming language, the grammars have to be written for every programming language for which code should be evolved for. But this process has to be done only once, as the grammar can be reused because of the aim for tackling general program synthesis problems. The grammars written for this thesis target Python and are available online [37] as part of the G3P plugin used for all experiments as well as in Appendix B, which also describes how the automatic combination of grammars takes place.

4.2.2 Skeleton

The skeleton is the second part of the grammar design approach. The skeleton consists of multiple components. It contains the libraries used, if any, required by the grammars and the evolved code, protected methods to guarantee evaluation safety, a method header for the code that should be evolved and a fitness function. A shortened example of a skeleton is shown in Figure 4.3 and its parts are highlighted. While the libraries and protected methods are mostly problem independent, the method header and fitness function are specific to a problem. GP systems usually have protected methods in regression, like protected division to avoid division by zero. Protected methods are defined in the skeleton and are used in the grammar to prevent exceptions during runtime. The protected methods can be reused across different problems. The method header specifies the input and output variables for the problem. The variables are defined in the

```
import maths
import Levenshtein

# Protected Methods
def div(nom, denom):
  return num if denom <= 0.00001 else num / denom
...

#  evolved function
def evolve(in0, in1):
  <insert_code_here>
  return res

def fitness(in_val, out_val):
  fit = []
  for (i, o) in zip(in_val, out_val):
    fit.append(abs(evolve(i[0], i[1]) - o[0]))
  return fit
```

Figure 4.3: Example skeleton in Python. Available libraries are highlighted in green, protected methods in blue, method header in red and fitness function in orange.

grammar with the corresponding data type. In case of Python, the data type does not have to be specified in the method header. The skeleton even specifies the fitness function, which provides certain benefits. As the whole skeleton is in the same programming language as the code that is going to be evolved, it can be executed as a whole with the evolved code. There is no need to implement every fitness function for every program synthesis problem within the GP system as it will be different for every single one. Additionally, the grammars and skeleton can be easily exchanged with other researchers or practitioners. A G3P system just needs to be able to execute code of the target programming language.

4.2.3 Comparison of Program Synthesis Approaches

The design pattern approach presented in this chapter poses many advantages over the methods discussed in Section 4.1. The main advantage over previously used grammar-based approaches is that this grammar design pattern produces reusable grammars which are not customised to a single specific problem. The

design pattern is also not tailored for a particular programming language, like PushGP. Grammars for arbitrary languages can be created, which has to be done only once, and the full extent of libraries available in a language can be integrated if desired as the evolved code is executed in the language it is targeted for. There is also no need to reimplement a system as with SFGP if one does not want to use the original source because grammars can be exchanged between any G3P system as well as the skeletons. The grammar design pattern approach also satisfies the closure property due to the type consistency in grammars and evaluation safety from the skeleton. This helps the GP system to work effectively and avoids syntactically incorrect programs as well as should keep errors to a minimum. The evolved code can also be integrated into real-world applications as the code is not evolved in some form of pseudocode.

Although the presented grammar design pattern offers advantages over other systems, it also has its limitations. One disadvantage is that grammars and skeletons have to be created by hand for every new language. Even though that requires substantial knowledge of the new programming language, especially to make the grammars type consistent, this process has to be done only once for each programming language. Afterwards, only the problem specific parts of a skeleton have to be adapted to specify the correct method header for a problem and to have the correct fitness function. An advantage PushGP still has is that no variables have to be defined as data is stored on stacks. Grammars require the number of variables available beforehand and have to be estimated per problem. Although it is possible to adapt grammars during the search process [130], this solution has not yet been further investigated for the grammar design pattern approach proposed here. One benefit bespoke grammars have over more general grammars is that the search space is most likely smaller. Even though this might be the case, as grammars can be easily be adapted, the search space can be further restricted in the proposed approach as well. Additionally, writing a bespoke grammar for a specific problem can be time intensive, while the grammar design approach can be applied to arbitrary problems, as will be shown in Section 4.3.

The advantages and disadvantages of the grammar design approach are summarised below:

Pros:

- All individuals are syntactically correct

- Protected methods keep exceptions to a minimum
- Produces code in a programming languages used by practitioners
- Use of libraries of programming languages
- Grammars and skeletons are reusable for arbitrary problems

Cons:

- Grammar and skeleton have to be created by hand and for the first time for a new language
- Grammars have to be made type consistent
- Variables have to be predefined
- Search space might be more extensive than required

As code for Python is going to be evolved in the following experiments, grammars for all data types from Figure 4.1 and skeletons for the problems of the general program synthesis benchmark suite, see Section 2.3.2, are already available in Python. Although the process of writing the grammars from scratch can seem complicated, extending them with additional functionality is easy. Additionally, the Python grammars can also be used as examples for other programming languages.

4.2.4 Python Specific Differences

Even though the concept presented can be applied to create grammars for arbitrary programming languages, some parts are specific to a language. The main Python specific difference in the grammars, compared to other languages, is that Python uses indentation instead of brackets to separate code blocks. It is not possible to add information about the code block level, the number of indentations required, within a context-free grammar, as information of the indentation level of the previous block would be needed, also known as off-side rule [131]. A Context-Sensitive Grammar (CSG) [87] would be required to cope with indentation levels. To avoid CSG's and at the same time keep it similar to other languages, special characters "{:" and ":}" have been introduced in the grammar to identify code

blocks. Before executing the evolved code, the code created by the grammar will be preprocessed to add the indentation required and to format it according to the Python syntax.

Another aspect that has to be taken care of that affects all languages is that methods used in different languages have different behaviour. Therefore, protected methods need to be adapted not only due to the different syntax but also due to the default behaviour of a language. Protected methods are necessary, as mentioned before, to guarantee evaluation safety and keep exceptions which result in invalid individuals to a minimum.

4.2.5 Invalid Individuals

While grammars guarantee syntactically correct code and protected methods in the skeleton reduce errors, runtime exceptions and long evaluations cannot be avoided entirely. If any runtime exceptions occur, for example memory overflow or stack overflow, the individual will be marked as invalid and assigned the worst possible fitness. The main problem still is long evaluation and the halting problem [132, 133]. The halting problem is that given a program and an input, it is not possible to prove if the program will stop eventually or run forever, for example due to infinite loops. In most cases in GP, this is not a major concern as programs can be given a time budget within programs have to finish. The idea proposed to let G3P run real code within the corresponding interpreter or on the machine directly can be executed in a separated process which can be stopped if not finished within the time budget or even a virtual machine could be used to have even more control. If a program needs to be stopped, its corresponding individual will be given the worst possible fitness as no evaluation was possible. A timeout parameter for the time budget can be given to the G3P system. This parameter should not be seen as a way to control for code performance, but as a last resort to stop evaluation. One origin for problems with programs that do not stop was found during preliminary experiments which are loops. Loops are necessary in programming to execute the same set of instructions multiple times. Loops may never stop if its stopping condition never becomes true. To reduce the number of individuals that timeout which results in the worst possible fitness and no information about what the individual does can be gained, a maximum number of loop iterations is added. This can be done by adding a small piece of code at

the end of the loop statement in the grammar, that always stops loops after a certain number of iterations. The number of iterations allowed is set for all loops globally, which means that the iterations of all loops are added up. The reason is that even a small number of iterations for each individual loop can result in long evaluations if the loops are nested which increases the runtime exponentially.

4.3 Experimental Setup

The general program synthesis benchmark suite presented in Section 2.3.2 has been used to test the grammar design pattern presented in this chapter, as it contains an extensive set of 29 program synthesis problems of varying difficulty. The benchmark suite has been tested on first with PushGP [95], which will, therefore, be compared to. In order for a fair comparison, the functions available in the grammars are only a subset of the built-in functionality available in Python. The parameter settings for the experiments are as close as possible to the settings give in [1], but most of them are PushGP specific. PushGP has been tested with three different selection operators. Tournament selection, Implicit Fitness Sharing [134] and Lexicase selection [61]. Experiments conducted to compare to PushGP will be executed with tournament selection and lexicase selection. Tournament selection will be used, because it is the most commonly used in GP, and lexicase selection because it has been the most successful one on program synthesis problems. Other parameter settings have chosen from typical GP settings and are summarised in Table 4.1. Additional to the required input and output variables for each problem, a number of three additional variables per data type is used to store temporary data, which is more than enough for all problems. This number has to be predefined as explained in Section 4.2.1. Increasing this number will also increase the search space. The maximum execution time which is required to stop non-halting programs has been set to one second which is sufficient as the programs should be able to finish within less than a tenth of a second on each problem. PushGP uses a number of steps the interpreter is allowed to execute as it has full control of the interpreter. Using a number of steps is not possible when using default interpreters or compilers as such options are not available. The interpreter used in for the experiments is CPython, which is the reference implementation of Python.

Table 4.1: Experimental parameter settings

Parameter	Setting
Runs	100
Generations	300 [1]
Population size	1000
Tournament size	7
Crossover probability	0.9
Mutation probability	0.05
Elite size	1
Node limit	250
Variables per type	3
Max execution time	1 second

[1] 200 Generations for Median, Number IO and Smallest as set in [1]

Although the experiments conducted are the same as in [128] the results may differ slightly. In [128] grammars have been combined by hand, which is now done automatically, which removes possible errors made by hand as well as unit-productions are removed. Lastly, minor tweaks to the grammars have been made since the original paper was published.

4.3.1 PushGP Differences

PushGP and the grammar design pattern are different approaches to program synthesis, and two critical aspects should be noted. In PushGP, programs include instructions to print data which is sometimes expected as output. The code evolved with the grammar design approach are methods with only input and output variables and no additional print instructions. Results that should be printed have to be returned. Therefore, certain problem specific terminals, mostly strings, that are only used to format the output are not included in the grammars. Due to this difference and because the grammars do not allow data structures to contain different data types to guarantee type safety, the problem String Differences has been excluded from the benchmark suite, as it requires to either print or to have a data structure containing different data types. String Differences has not yet been solved with PushGP.

4.3.2 Derivation Tree Structures

Chapter 3 studied the difference of derivation tree structures created by grammars or specifically by their recursive rules. Although the conclusion was drawn that there are performance improvements when the non-recursive part of a recursive rule can be exchanged in one operation, no answer could be given if the overall structure, either list-like or tree-like, influences performance on the problem at hand. It was experimentally shown that this is not the case for sorting networks. This conclusion cannot be extended to other problems. To this end, further experiments are conducted with the grammar design approach on program synthesis problems with different grammar structures. Similar to sorting networks, the recursive rules have been adapted to create different derivation tree structures. There are three direct recursive rules in the grammars for program synthesis. `<code>` creates a range of statements (`<statement>`). `<number>` and `<string_const_part>` create numerical and string constants. All three rules are part of structure.bnf. In Chapter 3, five different grammars have been created, although two of them only added an additional rule to aggregate the non-recursive part. From a perspective of different structures in the derivation tree, Chapter 3 only contained three different grammars. The recursive rules that created the different structures are shown in Table 4.2. Whereas the first rule *List* creates a list-like structure, the other two *Tree* and *Binary* create tree-like structures. The recursive rules, `<code>`, `<number>` and `<string_const_part>`, in structure.bnf of the grammar design pattern approach are replaced with the generic rules from Table 4.2 and each will be executed on the program synthesis benchmark suite as in the previous experiment, which already uses the *List* rule. These experiments aim to check if the structure of the derivation tree influences the performance of the search algorithm.

Other parameter settings are the same as before, shown in Table 4.1. Only lexicase selection is used for the experiments on grammar design, as the previous experiment will show that this selection operator is superior to tournament selection on program synthesis problems.

Table 4.2: Three different recursive rules for creating different derivation tree structures.

Name	BNF
List:	`<recursive> ::= <recursive><non-recursive>` `| <non-recursive>`

Tree:	`<recursive> ::= <recursive><non-recursive>`
	`| <non-recursive><recursive>`
	`| <recursive><non-recursive><recursive>`
	`| <non-recursive>`

Binary:	`<recursive> ::= <recursive><recursive> | <non-recursive>`

4.4 Results

This section analyses and discusses the results of the experiments conducted in this chapter. Even though the results are going to be compared to the results achieved with PushGP [1], the overall goal is to test if the grammar design pattern can solve general program synthesis problems without being tailored to a specific problem and therefore is flexible for use in the program synthesis domain in general. Additionally, advantages and disadvantages are uncovered during the analysis which can help to improve the concept further. Furthermore, the influence of the derivation tree structure on program synthesis problems is analysed as a continuation of the work in Chapter 3.

Although Helmuth et al. state in [135] that PushGP has been able to solve Checksum in contrary to [1] and that Vector Average has been solved more often, no actual numbers of how often this occurred have been given. Therefore, the numbers available in [1] will be used for comparison between G3P and PushGP.

4.4.1 Tournament Selection

The left-hand side of Table 4.3 shows the results achieved with tournament selection with G3P compared to PushGP. The values are the number of runs that produced a successful solution. A successful solution or correct individual is a program that was able to correctly solve all training cases and generalise to the unseen test cases. The grammar design approach with G3P was able to solve 11 out of the 28 problems at least once with tournament selection and even two, For Loop Index and Grade, that PushGP did not solve. All these problems require different data types including the list data structure. G3P solved a variety of

problems in terms of required data types without using tailored grammars. This shows that the approach presented in this chapter can be used to solve general program synthesis problems.

Nonetheless, PushGP found at least one correct for 13 problems including four that G3P was not able to solve with tournament selection. When looking at the numbers of runs that evolved a successful solution, then it can be seen that G3P in cases when a correct solution was found this number was often substantially higher. To be exact, for five problems, Median, Negative To Zero, Number IO, Smallest and String Length Backwards, G3P had at least ten more runs than PushGP that found correct solutions, while that was only the case for two, Mirror Image and Vector Average, the other way around.

4.4.2 Lexicase Selection

Using lexicase selection with G3P increases the number of problems solved at least once and the number of successful solutions found in 100 runs compared to using tournament selection, similar to the findings in [1] with PushGP. To the author's knowledge, this is the first time that lexicase selection was used in a G3P system, including [128]. With lexicase selection, G3P solves 16 problems at least once, which is five more than before and the number of successful solutions increased or is equal to tournament selection in all, except three cases. This shows that lexicase selection is a useful operator in the program synthesis domain. Further, the results indicate even stronger than with just tournament selection that the grammar design approach can be used to successfully tackle general program synthesis problems, which was the goal of this chapter.

When comparing G3P to PushGP on lexicase selection, PushGP is still able to solve more problems at least once. This is not surprising as the grammars created for G3P only contain basic Python functionality which was restricted not to exceed the functionality of Push. It should be pointed out to the reader that this is the case for all available data types except, string. Four functions, namely `strip`, `lstrip`, `rstrip` and `capitalize` have been overlook before the experiments and are part of the grammars, but not available in Push. These methods did not give G3P much, if any, advantage, as it performed poorly on problems that use the string data type, as will be shown in Table 6.1 in Chapter 6.

Table 4.3: Number of times a correct individual was found that solves all test cases for all 29 Problems with tournament and lexicase selection. PushGP results taken from [1]. The best results are marked in bold. The column "Diff" shows the difference between G3P and PushGP.

	Tournament			Lexicase		
Problem	G3P	PushGP	Diff	G3P	PushGP	Diff
Checksum	0	0	0	0	0	0
Collatz Numbers	0	0	0	0	0	0
Compare String Lengths	2	3	-1	2	**7**	-5
Count Odds	0	0	0	**12**	8	+4
Digits	0	0	0	0	**7**	-7
Double Letters	0	0	0	0	**6**	-6
Even Squares	0	0	0	1	**2**	-1
For Loop Index	1	0	+1	**8**	1	+7
Grade	4	0	+4	**31**	4	+27
Last Index of Zero	2	8	-6	**22**	21	+1
Median	48	7	+41	**79**	45	+34
Mirror Image	0	46	-46	0	**78**	-78
Negative To Zero	**79**	10	+69	63	45	+18
Number IO	94	68	+26	94	**98**	-4
Pig Latin	0	0	0	0	0	0
Replace Space with Newline	0	8	-8	0	**51**	-51
Scrabble Score	0	0	0	1	**2**	-1
Small Or Large	**8**	3	+5	6	5	+1
Smallest	**97**	75	+22	94	81	+13
String Lengths Backwards	18	7	+11	**69**	66	+3
Sum of Squares	0	2	-2	3	**6**	-3
Super Anagrams	0	0	0	0	0	0
Syllables	0	1	-1	0	**18**	-18
Vector Average	1	14	-13	**16**	**16**	0
Vectors Summed	0	0	0	**91**	1	+90
Wallis Pi	0	0	0	0	0	0
Word Stats	0	0	0	0	0	0
X-Word Lines	0	0	0	0	**8**	-8

Table 4.4: Statistics of how many problems have been solved, how often G3P or PushGP did better than the other with the same selection method and the average rank of each method.

	G3P		PushGP	
	Tour	Lex	Tour	Lex
# of solved Problems	11	16	13	22
Better than same selection method	9	10	7	11
Average rank	2.21	1.64	2.46	1.50

At the same time, PushGP includes some functionality that has not been added to the grammars, because the grammars should be kept as general as possible to contain merely built-in Python functions. PushGP was able to solve six problems at least once that G3P was not able to solve at all. While PushGP even has 11 problems where more successful solutions have been found, only 5 when excluding the ones G3P was not able to find any successful solutions, G3P outperforms PushGP on ten problems in this regard. And when looking at the difference of the number of successful solutions found, G3P has five problems, namely Grade, Median, Negative To Zero, Smallest and Vectors Summed, with a difference of 10 or more successful runs, whereas PushGP only has 3, Mirror Image, Replace Space with Newline and Vector Average. Of these three, G3P was not able to solve any of them. All this indicates that if G3P was able to solve more problems from the benchmark suite, it might outperform PushGP.

Table 4.4 shows the number of problems solved with each approach and selection operator, as well as the number of problems where an approach with a particular selection method outperformed the other with the same selection method. Finally, it shows the average rank of each method and selection operator. The average rank for lexicase selection is lower than with tournament selection, which indicates that lexicase selection outperforms tournament selection. Otherwise, both systems, G3P and PushGP, achieve similar ranks.

Overall G3P is able to solve a variety of different general program synthesis problems and achieve competitive results to PushGP, the only other GP system that has been tested on this benchmark suite as well. In a study on the difficulty of benchmarking program synthesis methods [136], two other non-GP systems that have been tested on the benchmark suite, namely Flash Fill [81] and MagicHaskeller [9, 80]. Although it should be mentioned that Flash Fill's primary purpose is not to generate general-purpose programs but to be used within Mi-

crosoft Excel to perform string manipulation and therefore could not be tested on all benchmark problems, both systems were outperformed by G3P and PushGP. Flash Fill could not find a solution to any of the 21 problems it has been tested on and MagicHaskeller only to 6 out of all 29 problems. Hence, these two systems have not been used in the comparison of this chapter.

4.4.3 Generational Progress and Invalids

For the experiments conducted in this chapter, the parameter settings have been taken from [1], which use 300 generations for all experiments except Number IO, Smallest and Median. Table 4.5 shows the average generation when a solution was found for each problem with G3P with lexicase selection. If a run did not produce a correct solution, it has been excluded. This number is below 100 for ten problems, which means that solutions have been found in the first third (or half) of the search. Only 33 correct solutions over the total of 592 solutions found for all problems have been discovered after generation 200. This indicates that in some cases the number of generations, therefore computational effort, can drastically be reduced without significantly reducing the number of successful solutions found. No numbers for PushGP are available to compare this to.

Another phenomenon that was also present for PushGP is that for certain problems a high number of solutions were found which correctly solved all training cases but failed to generalise to the test cases. G3P shows similar behaviour. G3P produced on seven problems more than 20 programs that correctly solve training but not test. In case of Compare String Lengths this number is even 97 and for Grade, Mirror Image and Small Or Large 50 or higher. This phenomenon might be caused by lexicase selection, as it also appears with PushGP but does not happen that regularly with tournament selection.

The skeleton used in the grammar design approach contains protected methods to provide evaluation safety, and the grammars provide type safety. The idea behind this is to avoid exceptions which marks them as invalid and results in giving individuals the worst possible fitness. This helps to have an effective search process. Exceptions can still occur, e.g. due to memory errors or stack overflows. Also if an evaluation takes longer than the maximum execution time of one second, it will just timeout and also be marked as invalid. Table 4.5 show the percentage of invalid individuals encountered during the search averaged over all runs per

Table 4.5: Average generation a solution was found with G3P with lexicase selection and the percentage of invalids including (incl.) and excluding (excl.) individuals that timed out during evaluation.

Problem name	Avg. generation	Invalids incl.	Invalids excl.
Checksum	-	0.30%	0.001%
Collatz Numbers	-	0.10%	0.001%
Compare String Lengths	11.50	0.37%	0.002%
Count Odds	176.25	1.46%	0.003%
Digits	-	1.41%	0.018%
Double Letters	-	0.68%	0.037%
Even Squares	262.00	1.10%	0.007%
For Loop Index	149.13	1.36%	0.002%
Grade	91.10	1.14%	0.001%
Last Index of Zero	88.59	2.36%	0.006%
Median	13.51	0.23%	0.000%
Mirror Image	-	0.84%	0.000%
Negative To Zero	35.32	2.48%	0.014%
Number IO	45.18	0.00%	0.000%
Pig Latin	-	0.58%	0.030%
Replace Space with Newline	-	0.25%	0.013%
Scrabble Score	179.00	0.71%	0.007%
Small Or Large	77.00	0.63%	0.000%
Smallest	6.78	0.69%	0.000%
String Lengths Backwards	81.17	1.10%	0.005%
Sum of Squares	245.33	0.16%	0.002%
Super Anagrams	-	0.98%	0.000%
Syllables	-	0.39%	0.003%
Vector Average	173.81	0.55%	0.031%
Vectors Summed	42.16	1.51%	0.013%
Wallis Pi	-	0.15%	0.000%
Word Stats	-	0.61%	0.009%
X-Word Lines	-	0.90%	0.014%

problem including and excluding invalids due to timeouts. As can be seen, most problems produce less than one percent of invalids and only two problems, Last Index of Zero and Negative To Zero produce more than two percent. Most of these invalids happen due to a timeout during the evaluation. All problems produce invalids of less than 0.04% due to other exceptions. This shows that the skeleton provides evaluation safety, although further protections could be put in place by restricting numbers to a specific range, as Python does not have a real upper or lower limit, or the length of strings and lists which can cause memory errors.

4.4.4 Derivation Tree Structures

Finally, Table 4.6 shows the number of successful solutions found over 100 runs with G3P and lexicase selection for the three derivation tree structures presented in Section 4.3.2 to analyse if the derivation tree structures influence the search performance. The results of List are the same as in Table 4.3 as the same structure has been used. Overall, it can be seen that all three derivation tree structures achieve similar results on all problems. On the one hand, Binary is the only on that successfully solved Mirror Image once, but List was able to produce 51 solutions that solved all training cases for Mirror Image, but they did not generalise to the test. On the other hand, Binary does not have any solutions for Even Squares and Sum of Squares. The number of found solutions is rather small for all three problems with any derivation tree structure. Even when examining the p-values, which have been calculated with the Wilcoxon Rank sum test and shows if there is a significant difference between two structures used, there is no evidence to support that one derivation tree structure is superior to another. This indicates that the derivation tree structure of grammars has little influence on the grammar design on general problem synthesis problems as was also demonstrated for sorting networks in Chapter 3.

4.5 Summary

In this chapter, a grammar design pattern for creating grammars in arbitrary programming languages to tackle general program synthesis problems has been presented. The design approach consists of multiple grammars of various types to provide type safety and a skeleton to achieve evaluation safety. This approach

Table 4.6: The Table shows the number of successful solutions found over 100 runs with different derivation tree structures as well as the p-values when comparing the test fitness of two structures calculated with the Wilcoxon Rank sum test. p-values lower than 0.05 are marked in bold (L = List, B = Binary, T = Tree).

	Solved Runs			p-value		
Problem	List	Tree	Binary	L-T	L-B	T-B
Checksum	0	0	0	0.877	0.812	0.898
Collatz Numbers	0	0	0	0.521	0.310	0.690
Compare String Lengths	2	4	1	0.138	**0.036**	0.686
Count Odds	12	12	11	0.399	0.660	0.166
Digits	0	0	0	**0.022**	0.102	0.553
Double Letters	0	0	0	0.833	0.248	0.187
Even Squares	1	1	0	**0.000**	**0.009**	**0.027**
For Loop Index	8	13	11	0.915	0.491	0.690
Grade	31	34	33	0.738	0.652	0.812
Last Index of Zero	22	29	29	0.349	0.573	0.679
Median	79	90	89	**0.029**	0.063	0.781
Mirror Image	0	0	1	0.602	0.388	0.224
Negative To Zero	63	72	76	0.102	**0.012**	0.424
Number IO	94	98	93	**0.006**	**0.007**	0.860
Pig Latin	0	0	0	0.210	0.931	0.217
Replace Space with Newline	0	0	0	0.391	0.719	0.621
Scrabble Score	1	2	4	0.238	0.492	0.106
Small Or Large	6	9	7	0.435	0.593	0.947
Smallest	94	88	93	0.146	0.828	0.212
String Lengths Backwards	69	61	61	0.354	0.102	0.443
Sum of Squares	3	3	0	**0.000**	**0.000**	0.549
Super Anagrams	0	0	0	0.067	0.249	0.487
Syllables	0	0	0	0.233	**0.000**	**0.000**
Vector Average	16	6	7	**0.000**	**0.031**	0.098
Vectors Summed	91	93	83	0.636	0.072	**0.026**
Wallis Pi	0	0	0	**0.031**	0.060	0.589
Word Stats	0	0	0	**0.016**	0.421	0.248
X-Word Lines	0	0	0	**0.021**	0.437	0.177

solves the problem of creating a new tailored grammar from scratch for every program synthesis program that is going to be tackled with a G3P system.

Experiments have been conducted on the general program synthesis benchmark suite to validate that it can be applied to various problems requiring multiple data types and to evaluate the design pattern approaches' performance. The results achieved with the design pattern has been compared to PushGP, the only other system that has been tested on the benchmark suite with substantial success. The overall results of the grammar design approach are competitive to PushGP and showed that lexicase selection is the superior selection method for program synthesis problems. In many cases, G3P only used up a fraction of the available number of generations to find a correct solution, while the percentage of invalid individuals was kept small due to the evaluation safety of the grammar design approach. Further experiments have been conducted to show that the derivation tree structure has little influence on the search performance similar to sorting networks in Chapter 3.

Randomly selected solutions that solve each problem with the grammar design approach are shown in Appendix E for visualization of the code that can be evolved.

Part III contains three chapters which are expansions to this one. First, Chapter 5 analyses the computational effort required to solve program synthesis problems and uncovers signals that help practitioners improve success rates. Then, Chapter 6 shows an example of how grammars from the grammar design approach can be extended and analyses what is missing to solve more problems of the benchmark suite with G3P. Finally, Chapter 7 explains the concept of semantics in program synthesis in detail and introduces novel semantic operators in the domain of program synthesis to improve success rates.

Part III

Extended Experimental Research

Chapter 5

Refining Computational Effort for Program Synthesis

The previous chapter, a grammar design pattern to tackle general program synthesis problems has been introduced. The grammar design pattern approach with Grammar-Guided Genetic Programming (G3P) has been able to solve a variety of problems from the general program synthesis benchmark suite as has PushGP before. While Genetic Programming is capable of finding correct solutions, the actual success rate per problem is low in many cases. The work that follows studies the computational effort required to solve program synthesis problems to check if an increase in computational effort can increase the success rates or if there are underlying problems that prevent G3P from achieving a higher success rate. The reason is that although problems in the benchmark suite are of varying difficulty, the computational effort specified in [1], is identical for almost all problems. To this end, a subset of problems from the benchmark suite is identified on which G3P performs poorly. An increased amount of computational effort is provided for these problems, and the performance of G3P is analysed. The results suggest that an increase of computational effort is helpful to solve more problems and a bigger training set is required to help solutions to generalize to the test set. This chapter is based on work from [137].

5.1 General Program Synthesis Benchmark Suite Remarks

When the general program synthesis benchmark suite was first presented, PushGP was able to solve 22 of all problems at least one out of 100 times, although success rates vary depending on the problem. Since then, the datasets of two problems, Checksum and Vector Average have been adapted with additional data, which made it possible for PushGP to solve Checksum and increase the success rate of Vector Average [135].

Subsequently, the benchmark suite has been tested with other systems. A grammar design approach with G3P was tested in Chapter 4, whose results were compared to PushGP. Although PushGP was able to find at least one solution to more problems, in general, the success rate on problems that G3P found solutions for was higher on most problems.

A more exhaustive study of different inductive program synthesis methods was conducted by Pantridge et al. [136]. The five methods tested are again the two GP systems PushGP and G3P as well as Flash Fill [81], MagicHaskeller [80, 9] and TerpreT [138], which have not been tested on the benchmark suite before. TerpreT is the only one that has not been used on the benchmark suite, as no implementation of it is publicly available. As FlashFill and MagicHaskeller are deterministic, a single run is sufficient to check if a solution can be found. FlashFill was designed for string manipulation within spreadsheets. Therefore it cannot be applied to all problems and subsequently fails to find solutions for many problems in the benchmark suite. MagicHaskeller was applied on all 29 problems and at least managed to find solutions for 6 of them. The results are not compared on success rates, but merely on the fact if a solution was found or not, which seems reasonable for deterministic algorithms, but bears little meaning for stochastic algorithms. A stochastic algorithm like a GP system that is run up to 100 times on a specific problem and only comes up with a single solution is not reliable and could have found a solution by chance. In such a case the problem can hardly be counted as solved.

Low success rates are problematic when comparing different stochastic approaches or operators, as it is difficult to distinguish if a new approach was more successful or if the increase in successful solutions found was by chance. This issue becomes even more problematic, if many solutions have been found on a

program synthesis problem that do not generalize to the test set, which has been shortly discussed in [1] for PushGP and has been a problem for the grammar design approach as well, which will be examined in this chapter as well.

5.2 Experimental Setup

As the goal of this work is to analyse and better understand the existing suite of benchmark problems, the problems have been categorized based on the ease with which they have been solved to date. Problems have been put into three categories depending on the success rate of G3P in 100 runs.

High Problems with more than 50 successful solutions

Medium Problems with more than 5 but below or equal to 50 successful solutions

Low Problems with less than or equal to 5 successful solutions

These thresholds have been chosen without a statistical measure and are open to discussion, but seemed adequate when looking at the success rates achieved.

The focus of this chapter is to check if an increase in computational effort can increase the success rates or if there are underlying problems that prevent G3P from achieving a higher success rate. At the same time the experiments should give a deeper understanding of why success rates are low in some instances. The subset of problems selected for this study is shown in Table 5.1. The problems marked with an X in column "Used" have been tackled in this study. All problems categorized with a medium success rate as well as all that have been solved at least once from the low success rate category are used in this study, as the idea is to see if and how the success rate increases. Additionally, Checksum has been added, because the training set changed since the introduction of the benchmark suite, Super Anagrams because the number of runs that solve the training set was high in previous experiments, as well as Mirror Images which has been solved one time in 100 runs in Chapter 4 with the binary grammar structure as shown in Table 4.6.

5.2.1 Parameters and Computational Effort

The general program synthesis benchmark suite [1] was first used with PushGP as explained in Section 2.3.2 and therefore parameter settings for PushGP are

Table 5.1: Number of solutions found that correctly solve the test data with 100 runs on the general program synthesis benchmark suite with G3P. The table also shows if a problem is used in this thesis and the number of training and test cases.

Problem	Successes	Used	Training	Test
Number IO	94		25	1000
Smallest	94		100	1000
Vectors Summed	91		150	1500
Median	79		100	1000
String Lengths Backwards	69		100	1000
Negative To Zero	63		200	2000
Grade	31	X	200	2000
Last Index of Zero	22	X	150	1000
Vector Average	16	X	100	1000
Count Odds	12	X	200	2000
For Loop Index	8	X	100	1000
Small Or Large	6	X	100	1000
Sum of Squares	3	X	50	50
Compare String Lengths	2	X	100	1000
Even Square	1	X	100	1000
Scrabble Score	1	X	200	1000
Checksum	0	X	100	1000
Collatz Number	0		200	2000
Digits	0		100	1000
Double Letters	0		100	1000
Mirror Image	0	X	100	1000
Pig Latin	0		200	1000
Replace Space with Newline	0		100	1000
Super Anagrams	0	X	200	2000
Syllables	0		100	1000
Wallis Pi	0		150	50
Word Stat	0		100	1000
X-Word Lines	0		150	2000

Table 5.2: Experimental Parameter Settings. Increased effort settings are marked in bold.

Parameter	**Increased effort** / Default
Runs	100
Generations	**600** / 300
Population size	**2000** / 1000
Crossover probability	0.9
Mutation probability	0.05
Elite size	1
Node limit	250
Variables per type	3
Max execution time	1 second

available, which are not applicable to all other GP systems, except, e.g. population size and number of generations. The population size and maximum number of generations in the benchmark suite description was set to 1000 and 300 respectively, except for Number IO, Median and Smallest, which only were allowed to use 200 generations. When G3P was tested on these problems, the same settings were used to be able to make a comparison between the two approaches.

To analyse if GP is unable to solve the selected problems more often or if the computational effort it is given is just too limited, the population size and the number of generations were doubled which quadruples the search effort. All other parameter settings have been taken from Chapter 4 and no parameter tuning has taken place to be consistent and to allow for cross-comparison. A summary of the settings is shown in Table 5.2. Lexicase selection [61] is used as it was shown to be superior to other selection operators in the program synthesis domain. Additional to the usual stopping criterion, the maximum number of generations, for GP, runs are also stopped as soon as a successful solution based on the training data is found, as runs cannot further improve the results without looking at additional data when all training cases are solved.

5.2.2 Larger Training Set

A subset set of problems marked in Table 5.1 has been used to analyse problems that show a high success rate on training, but fail to generalize to the test set. G3P achieved a twice as high or even higher success on training than on test on a number of problems with increased computational effort. Therefore, an

additional experiment with an increased training set size was carried out with problems that showed such a characteristic after increasing the computational effort. Section 5.3.4 explains and discusses that experiment and its results.

5.3 Results

This section discusses the results of the experiments run with increased computational effort, where the population size and number of generations have been doubled. First, the overall success rate and performance of the two parameter settings are compared. Afterwards, specific problems with the datasets and overfitting are analysed, which are addressed in a subsequent experiment.

5.3.1 Success Rates

The number of successful runs on all the problems tackled as well as the average test fitness of the best training individual per runs are shown in Table 5.3. The number of successful runs is of importance in program synthesis as a program that not always gives the correct answer might be of little use unless it can be repaired after the run. Nevertheless, the test fitness of the best training solution gives a good indication of how close to the optima a solution is. A Wilcoxon rank-sum test is used to compare the test fitness of the best training solutions. The p-values are also shown in Table 5.3.

Statistically significant different values are indicated in bold. As expected when increasing the computational effort, many problems show a significant difference, 8 out of 13. In some cases, like Compare String Lengths and Grade, it is not surprising that no difference is found. Even though the number of successful solutions was increased, the number of solutions that pass all training cases is already rather high, which indicates that the runs have finished. Therefore the eight significantly different problems with increased effort are an improvement over the default setting. In many cases, even the number of successfully found solutions has drastically improved. In some cases, this number was nearly doubled or, in the extreme case of Sum of Squares, increased by more than eight times what was achieved with the default setting.

In all cases, except Checksum, the number of solutions found that correctly solve training has increased, but on 10 out of 13 problems, the increased effort

Table 5.3: Results on benchmark problems running G3P 100 times on each problem with increased effort. The table contains the number of successful runs on test and training data, the average test fitness and the average percentage of solved training and test cases of the best solution found during training with the improvement over the default settings and the p-value from Wilcoxon rank-sum test on the average test fitness. The result is compared to the default setting. The differences are shown in brackets.

Problem Name	Test	Training	Avg Fitness (% Improv.)	Avg Solved Train.	Avg Solved Test	p-value
Checksum	0 (+0)	0 (+0)	31065.12 (+13.74%)	53.37% (+21.20)	30.69% (+18.49)	**0.0000**
Compare String Lengths	6 (+4)	100 (+1)	103.35 (+6.98%)	100.00% (+0.01)	89.67% (+0.78)	0.5220
Count Odds	22 (+10)	28 (+16)	3881.11 (+24.25%)	59.58% (+15.92)	44.80% (+15.15)	**0.0029**
Even Squares	2 (+1)	4 (+3)	1947381.31 (+9.24%)	6.60% (+3.83)	5.37% (+3.32)	**0.0000**
For Loop Index	21 (+13)	35 (+15)	2591911.15 (+46.33%)	44.85% (+16.63)	44.14% (+16.47)	**0.0000**
Grade	34 (+3)	97 (+16)	70.19 (+71.51%)	99.90% (+2.65)	98.60% (+3.20)	0.2906
Last Index of Zero	26 (+4)	64 (+10)	2707.18 (+5.47%)	89.83% (+5.27)	71.87% (+3.55)	0.6149
Mirror Image	1 (+1)	94 (+43)	299.93 (+11.07%)	99.92% (+0.99)	70.01% (+3.74)	**0.0280**
Scrabble Score	17 (+16)	23 (+22)	4787.01 (+26.80%)	45.51% (+26.52)	32.80% (+24.33)	**0.0001**
Small Or Large	7 (+1)	87 (+28)	563.64 (+4.87%)	99.80% (+2.62)	88.69% (+1.02)	0.5900
Sum of Squares	26 (+23)	32 (+29)	58560.96 (+77.65%)	46.46% (+35.66)	43.58% (+34.72)	**0.0000**
Super Anagrams	0 (+0)	98 (+56)	278.48 (–4.79%)	99.99% (+0.33)	86.08% (–0.64)	**0.0049**
Vector Average	18 (+2)	19 (+2)	237307.42 (–4.59%)	37.11% (+2.42)	36.05% (+2.62)	0.4693

setting increases the number of solutions that do not generalize. A problem that is discussed in Section 5.3.3.

5.3.2 Accumulated Successful Solutions Over Generations

A more fine-grained comparison between the default settings and the increased effort can be made with Figure 5.1, which shows the accumulated successful solutions that have been found for every problem over generations. If all the runs of a problem stop before reaching the maximum number of generations due to successful solutions on the training data, the line in the plot is also stopped to indicate how many generations the experiments ran at maximum. This is only the case for Compare String Length with the increased effort parameter setting.

When comparing the number of successful solutions at generation 300, the results of only doubling the population size can be compared, which only doubles the computational effort. All problems have more successful solutions at generation 300 with increased effort except Even Squares and Vector Average, which are both just one off. This indicates that an increase of computational effort by a factor of four might not be required. For most problems, the increased population size accounts for most of the improvements over the default settings. Only a few problems, like Count Odds, For Loop Index, Scrabble Score and Sum of Squares, seem to take advantage of the increased number of generations. Especially Sum of Squares is able to double the number of successful solutions after generation 300. It is difficult to make any definitive comments for Even Square, as only one successful solution was found with the default settings and even with increased effort, only two were found, which have been found after generation 300.

Another aspect that should be mentioned is that although the increased effort parameter setting was given a total budget of four times the computational effort, this is only the worst-case scenario where no solution is found. As shown in Figure 5.1, the increase of the population on its own provided better results in most cases and many runs stop before reaching the maximum number of generations. On average the last generation reached over all problems is generation 370 with double the population and generations. This is less than 2.5 times the total budget of computation effort compared to the default setting.

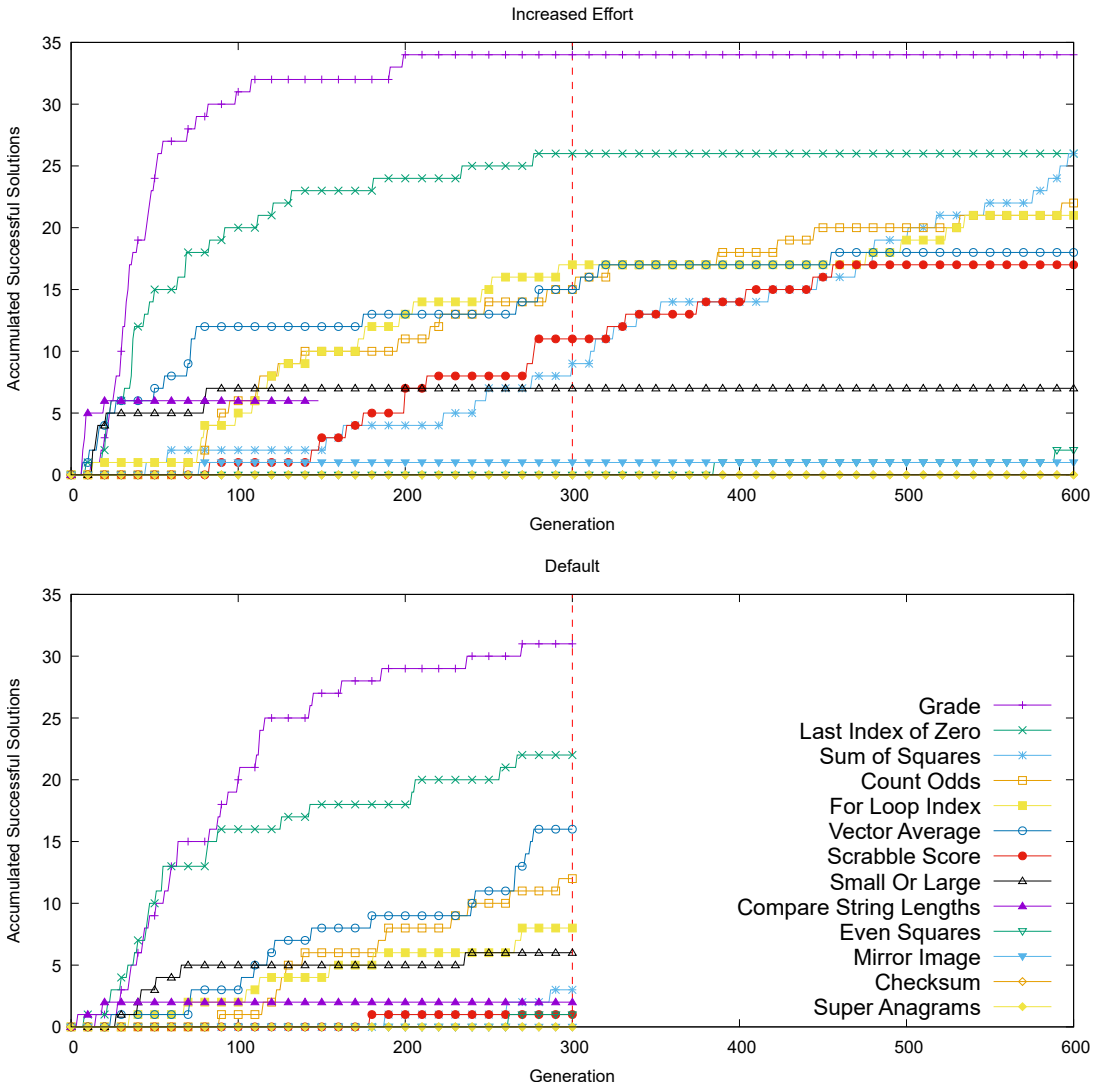


Figure 5.1: Comparison of accumulated successful solutions over generations over 100 runs. The top graphs shows the results with increased effort and the bottom one with the default settings. The dashed red line indicated generation 300, at which the default settings stop and the increased settings have used up twice the computational effort due to the increased population.

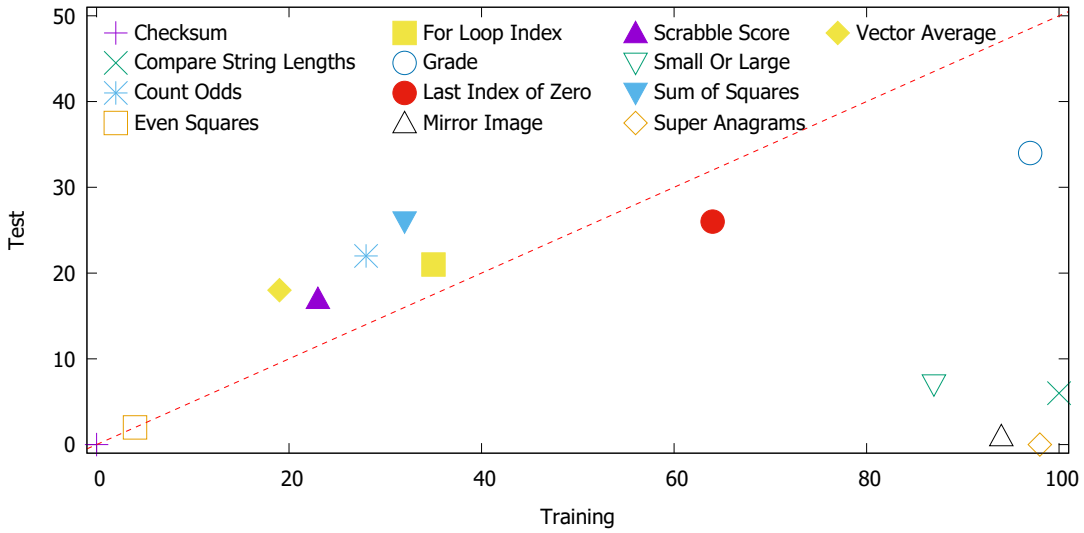


Figure 5.2: Number of runs which successfully solve the training and test set per problem. Note that the y-axis ends at 50 for visualization purposes. The red dashed line indicates at which point a problem is solved at least half the times on test as it is on training.

5.3.3 Problems with the Training Data

The increased effort further boosts the problem of having solutions that solve the training but do not generalize to the test set similar to the default parameter setting. This phenomenon has already been observed before with program synthesis problems [1]. This boost is expected, as increasing population and generations does not counter this problem, but shows that it is an even bigger concern. Figure 5.2 depicts the number of successful solutions on training and test. For five problems, Compare String Lengths, Grade, Mirror Image, Small Or Large and Super Anagrams, the training data is solved by almost all runs, but only a few or no solutions generalize to the test set. This may be due to overfitting or because the training data does not represent the problem well. All runs for the problems Compare String Lengths stop before reaching the maximum number of generations at generation 148 due to all runs finding solutions that solve the training cases. Even without increased effort 99 solutions that solve the training dataset have been found for Compare String Lengths.

As mentioned in Section 2.3.2, the datasets of two problems, Checksum and Vector Average have been adapted before, which lead to finding better solutions with PushGP [135]. This can be confirmed for Vector Average with the grammar

design approach with G3P [137]. This shows that the dataset has a tremendous influence on the success of the search and adapting the training set seems to be a logical conclusion to represent the problem more accurately and counter overfitting.

5.3.4 Larger Training Set

An additional experiment was carried out on all problems used in the previous experiment that have been solved more than twice as often on the training than on the test data. Compare String Lengths, Grade, Last Index of Zero, Mirror Image, Small Or Large and Super Anagrams. All problems that are below the red dashed line in Figure 5.2. Each problem gets an additional 100 training cases randomly generated from the same range as the original training set described in the general program synthesis benchmark suite. The experiment is run with increased effort as before.

The results are presented in Table 5.4 and compared to the increased effort setting with the original training set. Figure 5.3 illustrates these results. Three of the six cases, Compare String Lengths, Last Index of Zero and Small Or Large, show an increase of successful test solutions of up to 2.5 times. Only Grade and Mirror Image decrease slightly. Another positive side effect is that the number of solutions that solve the training but do not generalize to the test set decreases in all other cases. For Mirror Image and Super Anagrams, this number has decreased to less than half than before. The decrease shows the advantage of using a bigger training set, as it indicates that the previous training set might not have represented the problem space accurately. As a solution that solves training but not test is of little use and only stops the search prematurely. No solution found that solves training might be better than one that solves training but not test as it would continue the search. One should also be aware that just because a solution solves every case in the test set does not automatically mean that it is correct, as not every possible combination of inputs can be tested.

This experiment shows that with a bigger training dataset that better represents the problem, overfitting can be countered at least to some degree, as is expected in typical supervised learning. In most cases, the number of runs that only solve the training set is still double compared to the ones that generalize to

Table 5.4: Results of using an increased training data. The table shows the number of successful solutions for training and test. The difference to the increased effort setting with the original dataset is shown in brackets.

Problem Name	Test	Training
Compare String Lengths	12 (+6)	100 (+0)
Grade	29 (–5)	93 (–4)
Last Index of Zero	41 (+15)	79 (+15)
Mirror Image	0 (–1)	24 (–70)
Small Or Large	18 (+11)	88 (+1)
Super Anagrams	0 (+0)	45 (–53)

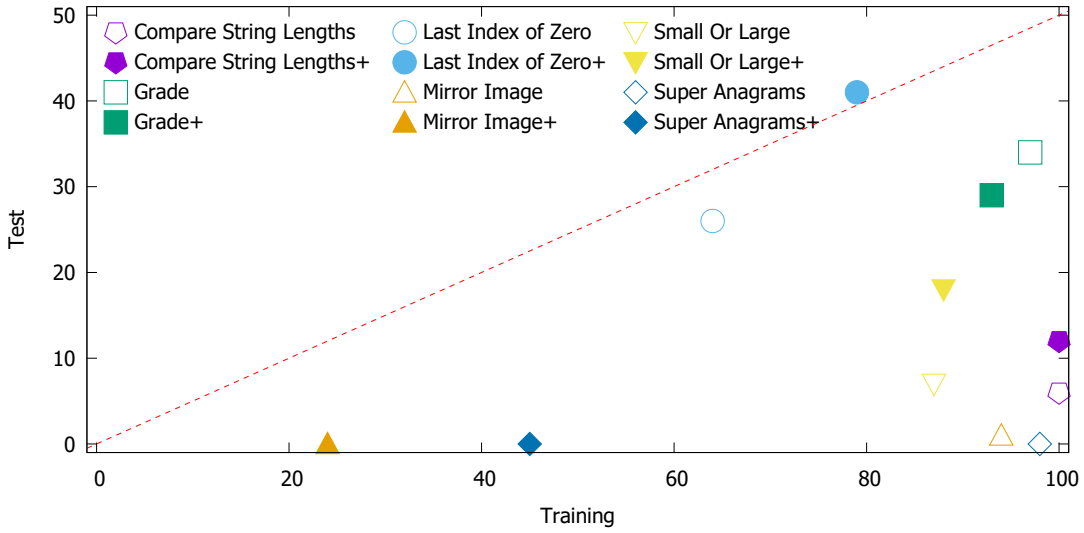


Figure 5.3: Number of runs which produce successful solutions that solve training and test with the larger training set. + marks the problems that were run with the increased training set. Note that the y-axis ends at 50 for visualization purposes. The red dashed line indicates at which point a problem is solved at least half the times on test as it is on training.

the test set. Further investigation is needed to understand the problem better and solve it.

One approach by Helmuth et al. [135] is a post simplification process that is shown to improve generalization of programs as well as that smaller programs tend to generalize better. Therefore, it might be worth to run experiments with smaller tree sizes.

5.4 Computational Effort Discussion

Regarding the problems tackled with G3P in this chapter, multiple signals have become apparent that hint at what steps should be taken to increase the success rate on program synthesis problems with G3P.

The first signal for problems with low success rate is the check if training was solved significantly more often then test, also referred to as overfitting. If that is the case, increasing the computational effort will have little effect. The step to take in this case may be to adapt the dataset to represent the problem accurately or adding more data if available. Getting more data or adjusting the dataset is not always possible for real-world problems or when problems are used for comparing different methods. As datasets of some problems in the benchmark suite used in this paper have already been adapted to improve performance before, it is undoubtedly of interest to see what is required to solve others that are still not solved.

The second signal is a low number of solutions that solve training and test. An increased population size has improved most problems. Additionally, an increased number of generations further improved the results of some problems and may be necessary if increasing the population size is not enough. It has not yet been identified what type of problems may benefit from the increased number of generations at this moment.

The improvement of success rates is an iterative process as increasing the computational effort can result in an increase of solutions that solve the training, but may fail test and adapting the dataset can lead to requiring more computational effort. If neither of those two methods improves the results, other steps have to be considered, like using better operators or adapting the search space, e.g. by changing the function set.

It should be noted that the first signal to counteract overfitting by increasing the training set to more accurately represent the problem at hand is just one measure to approach this problem. Although there is extensive research on the topic of overfitting [139, 140, 141, 142, 143, 144], more advanced techniques have not yet been regarded. Especially, one has to first check, what approaches are applicable to program synthesis. For example, early stopping [140, 141] is of no use, because stopping before all training cases are solved lead to a program that is not correct. The reason is that while solutions in domains like regression may be sufficient if they are approximations of the global optima, for many program synthesis problems, as is the case with the benchmark suite used in this work, a solution that nearly solves all cases is of little or no use. Nevertheless, considering research about overfitting from other domains could be beneficial for the program synthesis domain in future work.

5.5 Benchmark Suite Discussion

Although the general program synthesis benchmark suite provides a well-defined set of problems that helps researchers compare different approaches to program synthesis, the benchmark suite may require improvements, especially in regards adapting computational effort corresponding to the difficulty of a problem, to be of better use. At the same time modifications should not be made without considerations of other systems than G3P to avoid adjusting the problems to one system rather than addressing general issues.

As mentioned before, the computational effort, in terms of population size and number of generations, specified in [1] is the same for all problems, except for Median, Number IO and Smallest, which use a reduced number of generations. As shown in Section 5.3, G3P is able to significantly improve its results on a number of problems when the computational effort is increased. This shows that in many cases the computational effort restricts G3P. While other problems are fairly easy to solve as outlined in Section 5.2. Figure 5.4 shows the accumulated successful solutions over generations with default settings. All of these problems with a general high success rate have a steep increase in successful solutions found within the first 50 generations, except String Lengths Backwards. For three of these six problems, Median, Negative To Zero and Smallest, only a few more

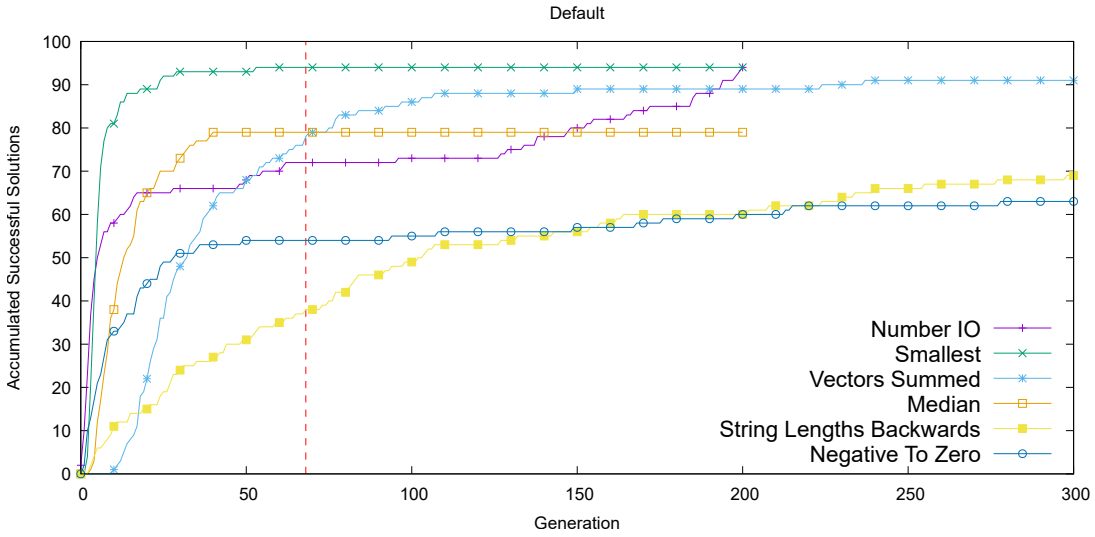


Figure 5.4: Accumulated successful solutions over generations over 100 runs with default settings. The dashed red line indicates that at generation 68 half the solutions on average over all problems have been found.

solutions are found after generation 50 and half of all solutions on average will be found before generation 68.

Adjusting the computational effort for problems in the general program synthesis benchmark suite could make them more useful for comparing different approaches. Increasing population size and number of generations for more difficult problems may yield higher success rates while reducing these numbers for easier problems could make them better benchmarks because problems that are solved every run and problems that are only solved once in a 100 runs barely provide useful information for comparisons between different approaches. To make proper adjustments to the computational effort, similar experiments as in this chapter should be run with other program synthesis systems, like PushGP.

5.6 Summary

The grammar design approach to tackle program synthesis problems has been investigated with an increased computational effort to analyse if the success rates on certain problems can be improved. To this end, a subset of problems with low and medium success rate has been selected and rerun with double the population size and number of generations to get insights on the amount of computational effort

required for each problem. While a statistically significant increase in performance could be verified, underlying problems with the general program synthesis benchmark suite could be identified. One is that the computational effort is not adjusted to the difficulty of the problems it contains, which could be useful for comparisons between systems. Also, the issue of overfitting problems, already observed in Chapter 4 and [1], drastically increases with increased computational effort and was addressed by increasing the training set size. A number of signals that can help practitioners to raise the success rates for program synthesis problems with G3P have been identified in Section 5.4.

The next chapter is the second expansion of Chapter 4 that is independent of what has been discussed above. The second expansion focuses on extending the grammars of the grammar design approach with additional functionality already available to other program synthesis systems, as well as recursion. Additionally, a grammar for characters has been added as many problems require characters as specified in [1], which technically is covered by the string grammar, but has been identified as one reason that G3P fails to solve particular problems.

Chapter 6

Extending Program Synthesis Grammars

The grammar design pattern approach with Grammar-Guided Genetic Programming (G3P) introduced in Chapter 4 is the most successful GP approach to tackle general program synthesis problems after PushGP. G3P achieved higher success rates on many problems from the general program synthesis benchmark suite than PushGP but failed to solve some problems at all. Current restrictions that could prevent G3P from solving additional problems are analysed in this chapter. These possible restrictions range from less "out of the box" functionality compared to PushGP within grammars, to not providing a *char* data type or recursion. The previously created grammars are extended with additional functionality and a new grammar for the data type *char* to counteract these problems. Extending the grammars to include functionality available in PushGP as well as a *char* data type allows for a fairer comparison between G3P and PushGP. The results show that these extensions provide G3P with the ability to solve more problems, especially ones that have not been solved before as well as one that has not even been solved with PushGP to date. This chapter is based on work from [145].

6.1 Grammar Design Approach Remarks

The functionality of the grammar design approach for program synthesis problems introduced in Chapter 4 has been kept to the basics of Python without including more than was available in PushGP. Therefore being a subset of Python and Push

functionality, except four string methods, namely `strip`, `lstrip`, `rstrip` and `capitalize`, which have been overlooked and are part of the program synthesis grammars, but are not available in Push. But these methods did not give G3P much, if any, advantage as G3P performed poorly on problems using strings as will be discussed below. Other data types do not exceed the functionality that is available in PushGP.

This has been done for two reasons. First, the grammar design approach was constructed to tackle general program synthesis problems without being tailored to a specific problem or problem set. Hence, the function set should not contain tailored functionality not provided by a language. Second, a comparison to another system can only be made if the functionality does not extend further than the other system. For example, adding the built-in *sum* function from Python would make solving the problem Vector Average fairly easy. Due to these restrictions, G3P might have been unable to solve certain problems in the general program synthesis benchmark suite detailed in Section 2.3.2.

The focus of this chapter lies in providing a fairer comparison between G3P and PushGP as well as determining how much better G3P can perform with functionality already available in Push especially on problems G3P has failed to provide solutions. To this end, differences between the function set of G3P and PushGP are identified, the problems G3P fails at in the general program synthesis benchmark suite are investigated, and grammars are extended accordingly. At the same time, the grammars shall stay as general as possible to be able to use them outside of the context of benchmark problems and should not be adjusted to "cheat" on any particular problem within the benchmark suite. Therefore, the functionality added to the grammars in this chapter is not allowed to be extended further than the function set available to PushGP. As the benchmark suite that has been used so far, proposes to have an explicit *char* data type which is currently missing in G3P [128] the possibility of adding it is further investigated.

Table 6.1 shows the results achieved with G3P with lexicase selection on the general program synthesis benchmark suite, which has been taken from Chapter 4. The table indicates that G3P with the current grammars is able to solve problems that require *string* as a data type, but it has difficulty to solve problems that require *char* as a data type as well. At the moment, only *string* is provided, because the initial grammars were based on Python which treats *char* as *string* of length one. The only problem G3P was able to solve that required *char* as

Table 6.1: Results of G3P on the general program synthesis benchmark suite sorted by successfully found solutions. *String* and *Char* row indicate if these data types have to be used when solving the problem according to [1].

	Number IO	*Smallest*	*Vectors Summed*	*Median*	*String Lengths Backwards*	*Negative To Zero*	*Grade*	*Last Index of Zero*	*Vector Average*	*Count Odds*	*For Loop Index*	*Small Or Large*	*Sum of Squares*	*Compare String Lengths*	*Even Square*	*Scrabble Score*	*Checksum*	*Collatz Number*	*Digits*	*Double Letters*	*Mirror Image*	*Pig Latin*	*Replace Space with Newline*	*Super Anagrams*	*Syllables*	*Wallis Pi*	*Word Stat*	*X-Word Lines*
Successes	94	94	91	79	69	63	31	22	16	12	8	6	3	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0
String					X		X					X		X		X	X		X	X		X	X	X	X		X	X
Char																X	X		X	X		X	X	X	X		X	X

data type was Scrabble Score. The reason might be because to solve this problem a string has to be iterated with a loop which iterates through every character, represented as a string of length one. Theoretically, all of the problems requiring *char* as a data type could be solved with the current Python grammars. While a programmer has no difficulty to understand how or when to use a single character string, it is more complicated for GP to find out how or when to use a string of length one. Adding a *char* data type could alleviate this problem and yield better results. An explicit *char* data type was mentioned in the description of the general program synthesis benchmark suite and available to PushGP as well.

6.2 Extending Program Synthesis Grammars

This section describes how the program synthesis grammars from Chapter 4 have been extended to include a *char* data type as well as additional functionality to have a fairer comparison to PushGP. Extending the grammar also means increasing the size of the search space as more programs can be generated from the grammar. Therefore, the extension of the grammars can also have a negative effect on the search performance.

6.2.1 Data Type Char

As shown in Section 6.1, G3P does poorly on problems that require a data type *char*. G3P only used *string* as it mainly relied on Python even though the concepts

can be applied to other languages as well and because a char can be interpreted as a string of length one. As many problems in the general program synthesis benchmark suite require to check or manipulate single characters, G3P not using a *char* grammar could explain why it currently fails at solving such problems. While programmers have the intrinsic knowledge that a string consists of characters, and a string of length one can be treated similar to a char, GP either has to discover this knowledge or has to be told a priori. The currently available grammar data types are *bool*, *integer*, *float* and *string*, as well as a list version grammar of each of these data types, plus the new *char* grammar. A list of char grammar is currently not included as the benchmark suite does not require it and strings can be viewed as a list of char. As G3P adds variables of the data types of every used grammar to the evolved program, including the char grammar makes it likely that chars are used as opposed to before where G3P had to find that a string of length one is required.

6.2.2 Recursion

Recursion is a method of programming where a program calls itself to solve a smaller instance of the same problem first and uses that solution to solve the initial problem. Recursion is a common strategy to tackle problems in GP [146, 147]. In many cases, a recursive solution can be significantly shorter in terms of code than an iterative program. PushGP is capable of evolving recursive programs and for a fair comparison should be part of the grammars for G3P as well.

A program needs to be able to call itself and a way to stop the recursion, usually an *if* condition called guard, to allow recursion. As the grammars of the grammar design approach for G3P are automatically merged depending on the required data types, and the number of input/output variables, as well as there types, a rule for a recursive call can be generated and added to the grammar. Figure 6.1 show the required rules in a grammar to add recursion. Figure 6.1a shows an example where `outputX` can be replaced with the correct type variable non-terminal (e.g. `<bool_var>`) and `inputX` with the correct type (e.g. `<bool>`). In a similar way, a return statement can be generated, as shown in Figure 6.1b, where every `resultX` is a variable containing a result value of the function.

The grammar used to define the control flow (*structure.bnf*) already contains *if* statements, but it is very likely that it might not be used and the program

```
<call> ::= 'if rec_counter < 900':{:
  'rec_counter += 1'
  <output1>','...','<outputN>' = evolve('<input1>','...','<inputN>')'
  'rec_counter -= 1'
':}'
```

(a) Grammar rule for a recursive call.

```
<call> ::= 'return result1, ..., resultN'
```

(b) Grammar rule for a return statement.

Figure 6.1: Rules required for recursion.

gets stuck in an infinite recursion and at some point will throw an error due to a stack overflow. A similar problem occurs with infinite loops as well and was handled by adding a guard to avoid any additional iterations if a certain limit is reached. A guard is used to avoid infinite recursion, see Figure 6.1a. The benefit of using this mechanism is that evolved programs will not throw an error and return a value. Therefore, the program will be given a fitness value based on what it returns instead of a default worst case fitness due to an error. This added guard merely stops the recursion to avoid infinite recursion but does not add any helpful information when to stop the recursion when evolving a program. G3P still has to evolve a guard to stop the recursion at the right time.

6.2.3 List Operations

When the grammars for program synthesis were introduced, grammars for lists of all data types were included but kept to the essential functionality. Items could be added at the end, inserted or replaced at a specific index or removed. Lists could be iterated, compared, checked if they are empty and their length could be determined as well as slicing of lists was possible. Any additional functionality the algorithm had to discover itself. PushGP offers more functionality out of the box, which has been added to the grammars for G3P, like reversing a list, counting the occurrences of an item, replacing or removing items if a condition is met etc. All of this functionality could be discovered as well. But for example O'Neill et al. [13] showed that GP has difficulties finding a solution to the integer sorting problem, however by adding a swap function the problem was easily solvable. As stated

before no further functionality has been added, that was not already available for PushGP as well. At the same time, it should be noted that adding additional functionality also increases the search space, which can make it more difficult to find a correct solution. While additional functionality can make it easier to solve one problem, it can make it more challenging to solve another. Therefore a decrease of successful solutions found on some problems is to be expected.

6.2.4 Additional Methods

Similar to the list operations in the previous section, additional methods were added to other data types that in general could have been discovered by G3P. One example that is also often not included for boolean problems is *XOR*, as it can be constructed with *AND*, *OR* and *NOT* and can make certain problems like multiplexer too easy [148]. To be able to have a better comparison between G3P and PushGP, such methods have been added as well. The extended grammars with the additional functions are provided in Appendix C as well as online [37]. The Push function set is outlined in [103]. Again, it should be noted that only functionality already available in PushGP has been added to the grammars.

6.3 Experimental Setup

The grammar design approach in a G3P system as presented in Chapter 4 is used with the extended grammars, which are described in Section 6.2 to tackle all problems from the general program synthesis benchmark suite (see Section 2.3.2) except Word Stats. A special focus is put on the problems that use the *char* data type, discussed in Section 6.1, as an additional grammar was added for those problems. The same parameter settings as in Chapter 4 are used for consistency and to allow cross comparison. The parameter settings are summarized in Table 6.2. Only lexicase selection is used as selection operator as it has shown that it produces a higher success rate than tournament selection in the program synthesis domain. Additionally to the stopping criterion of reaching the maximum number of generations, a run is stopped when it successfully solves the training, as no further improvement is possible.

Table 6.2: Experimental parameter settings

Parameter	Setting
Runs	100
Generations	300 [1]
Population size	1000
Crossover probability	0.9
Mutation probability	0.05
Elite size	1
Node limit	250
Variables per type	3
Max execution time	1 second

[1] 200 Generations for Median, Number IO and Smallest as set in [1]

6.4 Results

First, the overall performance of G3P with the extended grammars is compared to the initial grammars as well as the results of PushGP from [1]. Afterwards, the effect of the extended grammars on the search is analysed in more detail, primarily how the new *char* grammar and recursion have influenced the search. The results vary slightly from the ones in [145], as the experiments have been rerun due to minor grammar corrections, but overall the same conclusion can be drawn from the results.

6.4.1 Successful Solutions

Table 6.3 shows the solutions found for each problem with G3P with extended grammars for training and test with 100 runs. The results are compared to the previously achieved successful solutions with the initial grammars from Chapter 4. Only one, Scrabble Score, of the ten problems that require a *char* data type has been solved with G3P before. With extended grammars five problems have successfully been solved, namely Pig Latin, Replace Space with Newline, Scrabble Score, Syllables and X-Word Lines. Pig Latin has not even been solved to date with PushGP, to the best of the author's knowledge. Double Letters has not been solved with G3P, but one run discovered a program to succeeds in training but does not generalise. It is also interesting that Mirror Image, which has sometimes

been solved one time in 100 runs in previous experiments, has been solved 40 times, which might be due to the additional list operations.

The results also show that due to the increased search space, which is caused by the additional functions added to the grammars, the number of successful solutions decreases for some problems. Two problems, Even Squares and Vector Average, could not be solved anymore, but the success rate of the first one was rather small before as well. Furthermore, Table 6.3 includes the p-value for the Wilcoxon Rank sum test on the best test fitness of the two grammar approaches and shows a significant difference for nearly all of the problems. The significant differences are not surprising as the grammar has a massive influence on the search, as a function set has on vanilla GP. It should be noted that the significant difference is not always a positive effect as the success rates on some problems have decreased.

Finally, Table 6.3 shows the results of PushGP taken from [1] compared to G3P with extended grammars. According to [135], PushGP is able to solve Checksum after the original dataset has been changed. The comparison shows that both approaches have problems where one method is more capable of finding solutions than the other, but there does not seem to be a clear advantage over one or the other. Some problems have been solved with PushGP that have currently not been solved with G3P, but again the success rates of these problems are small, below 10, in most cases, which makes a comparison difficult. The low success rate is an issue that needs to be addressed by both approaches.

6.4.2 *Char* Analysis

This section analyses the usage of the *char* grammar. Ten problems use the *char* data type grammar. The grammar contains a rule `<char>` with productions for char variables, char constants and all functions that return a char value. Therefore, checking the percentage of nodes in individuals shows if GP is making use of the additional data type. Figure 6.2 depicts this usage. The percentage is calculated by summing the absolute amount of `<char>` nodes of all individuals of a generation in all runs divided by the absolute amount of all nodes of all individuals of a generation in all runs. It should be noted that the absolute amount of all nodes varies over time not only because individuals tend to grow over time, a maximum number of nodes limits this growth, but also because if a run finds a correct

Table 6.3: Successful solutions found with G3P with extended grammars on training and test with 100 runs as well as increase and decrease to the previously used grammars in brackets. The p-value shows if there is a significant difference in the best test performance between the two different grammars with 0.05 as level of significance. A significant difference is highlighted in bold. Finally, the results of PushGP on the benchmark suite from [1] and the difference to G3P with extended grammars in brackets are compared.

	G3P			PushGP
Problem Name	Test	Training	p-value	Test
Checksum	0 (+0)	0 (+0)	**0.0004**	0 (+0)
Collatz Numbers	0 (+0)	0 (+0)	0.2092	0 (+0)
Compare String Lengths	2 (+0)	96 (–3)	**0.0000**	7 (+5)
Count Odds	6 (–6)	7 (–5)	**0.0003**	8 (+2)
Digits	0 (+0)	0 (+0)	0.3589	7 (+7)
Double Letters	0 (+0)	1 (+1)	0.2163	6 (+6)
Even Squares	0 (–1)	0 (–1)	0.6381	2 (+2)
For Loop Index	8 (+0)	9 (–11)	**0.0005**	1 (–7)
Grade	24 (–7)	68 (–13)	**0.0038**	4 (–20)
Last Index of Zero	30 (+8)	96 (+42)	**0.0000**	21 (–9)
Median	46 (–33)	100 (+0)	**0.0000**	45 (–1)
Mirror Image	40 (+40)	94 (+43)	**0.0000**	78 (+38)
Negative To Zero	13 (–50)	29 (–37)	**0.0000**	45 (+32)
Number IO	77 (–17)	94 (–6)	0.6519	98 (+21)
Pig Latin	5 (+5)	7 (+7)	**0.0000**	0 (–5)
Replace Space with Newline	7 (+7)	22 (+22)	**0.0000**	51 (+44)
Scrabble Score	1 (+0)	1 (+0)	0.6060	2 (+1)
Small Or Large	5 (–1)	37 (–22)	0.0883	5 (+0)
Smallest	76 (–18)	100 (+0)	**0.0007**	81 (+5)
String Lengths Backwards	15 (–54)	16 (–56)	**0.0000**	66 (+51)
Sum of Squares	6 (+3)	6 (+3)	**0.0014**	6 (+0)
Super Anagrams	0 (+0)	56 (+14)	**0.0088**	0 (+0)
Syllables	32 (+32)	52 (+52)	**0.0000**	18 (–14)
Vector Average	0 (–16)	0 (–17)	**0.0001**	16 (+16)
Vectors Summed	32 (–59)	40 (–53)	**0.0000**	1 (–31)
Wallis Pi	0 (+0)	0 (+0)	0.1960	0 (+0)
Word Stats	0 (+0)	0 (+0)	**0.0167**	0 (+0)
X-Word Lines	1 (+1)	1 (+1)	**0.0000**	8 (+7)

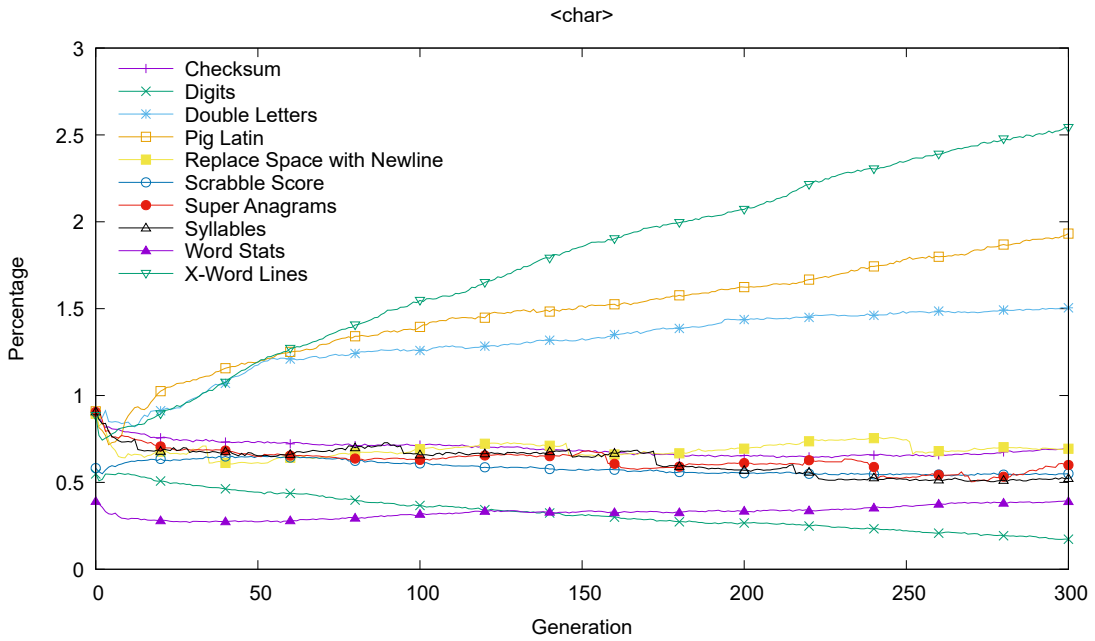


Figure 6.2: Percentage of `<char>` nodes in individuals averaged over 100 runs over generations.

solution, even if it is just on training, the run is stopped. If a run is stopped, the absolute number of all nodes changes, but also if a high number of certain nodes, e.g. `<char>`, is used in that particular run, the percentage of those nodes can suddenly drop.

In the initial generation, the percentage of nodes being `<char>` is nearly identical for some problems, which is expected as these problems require the same data types, which means the grammars are almost identical, except maybe input and output variables. Therefore, the grammars have the same structure and the same number of possible nodes, which leads to this effect. The percentage of `<char>` nodes used may seem small being between 0.5% and 1.5%, but considering the number of productions available in the grammar, it is rather high. In case of almost all problems, the usage of `<char>` nodes is either constant or increases over time, after a few generations. The only problem that seems to decrease the usage of `<char>` nodes is Digits. This decrease can be explained by how G3P is tackling the problems. While PushGP prints every integer for Digits, G3P has to return a list of integers as it does not use print statements and therefore does not necessarily need a char data type.

For some problems, especially Replace Space with Newline, Syllables and Super Anagrams, the lines are not as stable as for the other problems. The reason is that solutions that solve the problem at least for training have been found and runs are stopped as soon as this happens. Hence, the average percentage suddenly drops after a run is terminated. These drops indicate that runs that use `<char>` nodes more often seem to be able to find a successful solution earlier. This suggests that the *char* grammar improves the search for successful solutions.

6.4.3 Recursion Analysis

The percentage of recursion used can be checked in a similar way as in the previous section for *char*. Figure 6.3 depicts the percentage of recursion nodes, nodes that call the function that is being evolved, used over generations. The initial percentage is lower than with `<char>`, because there is only one recursion production rule in the grammar, whereas `<char>` is used by multiple functions. Afterwards, it drops even lower for all problems and is barely used overall. One exception is Smallest, which creates a spike before generation 50. This spike might be due to many runs finishing quite quickly and runs that do not use recursion for this problem finish earlier, in contrast to the smaller drops seen in Section 6.4.2.

As explained in Section 6.2.2, to use recursion, a method needs to be able to call itself and a stopping criterion. At the moment the GP system can evolve a method to call itself, but at the same time has to evolve a stopping criterion, which seems to make it too complicated to be used. Without a stopping criterion, the evolved program continues to call itself until the G3P system stops it. A way to improve this might be to adapt the grammar such that an *if* statement is added to the same production rule as the recursion. This ensures that the recursive call always adds a stopping criterion. The condition of the stopping criterion can be evolved by G3P. This could increase the chance to make G3P use recursion to solve problems.

6.5 Summary

The difficulties of solving multiple problems of the general program synthesis benchmark suite with a grammar design approach, introduced in Chapter 4, have been discussed. As some of this problems have been solved with another approach

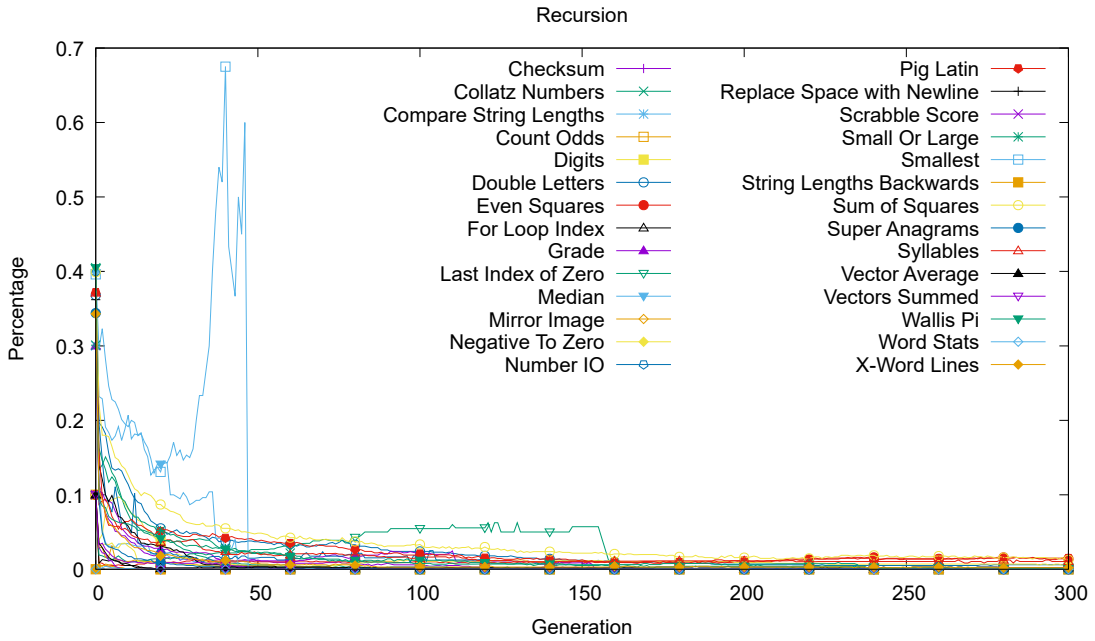


Figure 6.3: Percentage of recursion nodes in individuals averaged over 100 runs over generations.

before, the functionality of the grammars has been extended in various ways to be closer to previous methods, without "cheating" by adding functionality not used before. A significant enhancement of the grammars is that an explicit *char* grammar has been added as many problems operate on single characters instead of strings. Programmers are able to identify such characteristics of a problem quickly, while GP would have to discover such knowledge. As the benchmark suite proposes to use *char* as a data type, this additional information does not give G3P an unfair advantage when comparing to other systems and might be helpful when tackling problems outside of the benchmark suite.

Afterwards, the extended grammars are used to tackle the program synthesis benchmark suite, and the results are compared to the previously used grammars. The results show significant differences for nearly all problems and successful solutions have been found for previously unsolved problems with G3P. One problem, Pig Latin, has been successfully solved that was not solved by any other approach before. Additionally, a comparison with PushGP [1] has been made, as the extended grammars are closer in functionality to PushGP as before.

Due to the increased search space created by the extended grammars, a decrease of successful solutions found on previously solved problems was expected.

A way to dynamically adjust the functionality of grammars during runs could help avoid this problem, similar to the multi-level grammar approach by Saber et al. [130].

Code of solutions to problems that have not been solved in previous chapters have been put in Appendix E for visualization of the code that can be evolved.

The next chapter explains a novel concept of semantics in the program synthesis domain and introduces new operators based on the defined semantics, to create better offsprings by checking the behaviour of programs rather than making random syntactic changes.

Chapter 7

Semantics and Semantic Operators for Program Synthesis in Genetic Programming

Semantics is the behaviour or output of a program and this information has been used to improve search performance in GP [30, 34]. Semantic information has mainly been used in the regression and boolean problem domains [16], but to date is underutilised in program synthesis. The reason is that semantics is problem specific and therefore has to be defined for each problem domain. Hence, semantic measures require to be adapted and so do semantic operators. Defining semantics in program synthesis is especially difficult because program synthesis uses multiple data types and may even include data structures in contrast to regression and boolean problems, which only require real and boolean values respectively.

In this chapter, semantics for program synthesis will be defined in Section 7.2, which will be used to create novel semantic operators, crossover and mutation, in Section 7.3 and 7.4 that can use semantic information to improve the search performance. The behaviour of these new operators will be analysed on their performance as well as semantic aspects, like their ability to create semantically different children. While the initial semantic crossover, presented in Section 7.3, shows only minor improvements over conventional crossover, the insights gained have been used to propose enhanced semantic operators in Section 7.4 that significantly outperform standard operators. This chapter is based on work published in [149] and [150].

7.1 Semantics

Semantics and its most important properties are shortly recapped for the reader. A more detailed description of semantics and how semantics has been used in Genetic Programming can be found in Section 2.4.

Semantics can be defined as "the behavior of a program, once it is executed on a set of data" [16]. In the regression domain, a program is an arithmetic expression, which returns a vector of real values, when executed on data. Similarly, in the boolean domain, the semantics is a vector of boolean values. This semantic information can then be used in diverse ways to improve the performance of GP. Most approaches are based on new crossover [30, 31, 32, 33, 34] or mutation [35, 36] operators, but also semantic selection operators have been created [109, 108]. The downside of semantic operators, in most cases, is that as semantics are problem domain specific, operators based on semantics are domain specific as well. In contrast, conventional GP operators operate on a syntactic level. Hence they can be used regardless of the problem domain.

Two important properties of semantics that contribute to performance improvements are *semantic diversity* and *semantic locality* [34, 16]. While a high semantic diversity is necessary for covering the search space, semantic locality, which means a small change in a program corresponds to a small change in its semantics and therefore fitness, is essential for the search performance [34].

7.1.1 Semantic Operators

A variety of semantic operators has been introduced to GP and have shown to improve performance compared to conventional operators. The operators introduced in this chapter focuses on crossover and mutation, but also selection operators can harness semantic information [109, 108].

As stated previously, most research around semantics has been conducted in the regression and boolean problem domain. A series of operators have been introduced, some of which have been adapted and improved over time [32, 33]. One of which led to the Most Semantic Similarity based Crossover (MSSC) [34] in the regression domain. Operators introduced in this chapter are loosely based on MSSC. MSSC selects multiple pairs of subtrees, one from each parent, and chooses the pair that is most semantically similar. The most semantically similar pair is the one that has semantics, real-valued output vectors, whose mean absolute

difference is smallest compared to the other pairs. At the same time, the semantics of a subtree pair is not allowed to be equivalent, because if they were equivalent, the change would have no impact on the overall fitness.

Most semantic operators are based on the principle that the change that is being made has to change the semantics and if possible should be similar at the same time. Semantic mutation operators were created in a similar way, but instead of selecting a subtree from a second parent, it was generated at random, as in conventional mutation operators [35, 36].

7.2 Semantics in Program Synthesis

Semantics being domain specific means that it needs to be specified for each domain as well as the semantic operators need to be tweaked. For a regression problem, the semantics is a vector of real values. In the case of program synthesis, the output can be multiple vectors of different data types. A semantic similarity can be calculated on the difference between the variables of two programs. Similar to semantics in regression, it is not only possible to get the semantics of the final output, but also of intermediate steps. In the case of program synthesis, intermediate steps can be one or more executable statements. After every statement, the change of variables can be checked. Therefore, the semantics of every statement can be saved and used in a genetic operator. Many semantic operators exchange subtrees of a GP tree and need to be able to evaluate semantics of subtrees instead of a whole solution. Therefore, the first parent node representing an actual program statement is used to measure the semantics of a subtree which only represents part of a program statement (e.g. a binary comparison).

The process of logging variable changes in a program is called *tracing* and produces a *trace*. These traces are used to check the semantics of every statement in the program and to measure semantic similarity. A short example of a program, the corresponding GP tree and its trace is shown in Figure 7.1, which is used to explain how the semantics for program synthesis can be defined. The figure shows a short program that only consists of two statements. The GP tree is a derivation tree that is generated when using a grammar-based variant of GP. The bottom of the figure shows the variable settings at different states during the execution of the program referenced with numbers from `1*` to `3*`. It should be noted that in contrast to regression or boolean solutions, the output may not

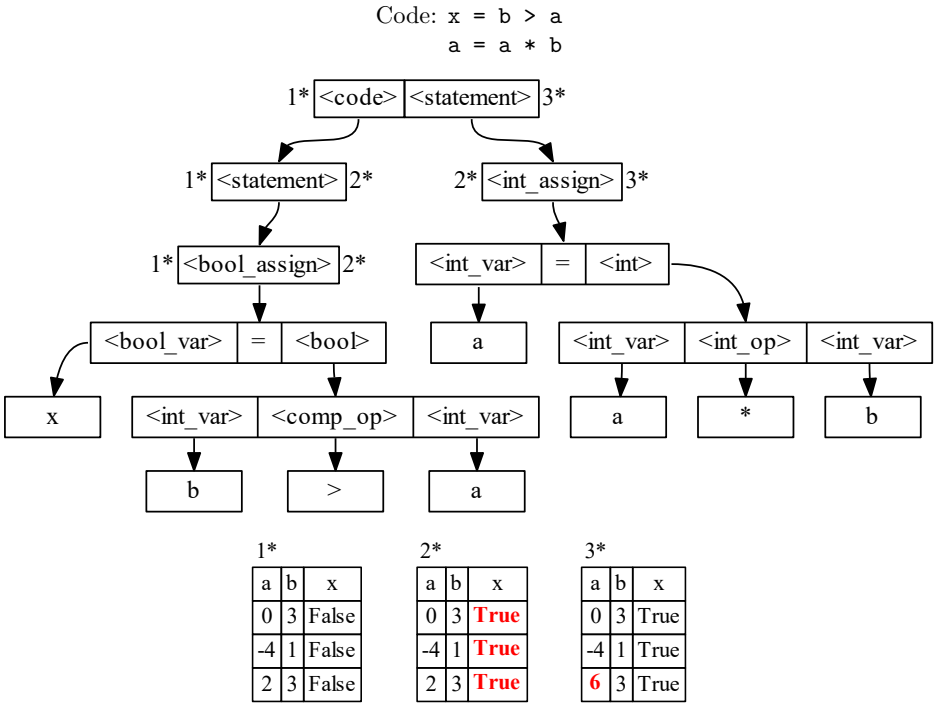


Figure 7.1: Program synthesis semantics example. A sample of code is shown with its corresponding derivation tree and its trace for three different inputs. The state of the variables before running the code is shown in variable setting `1*`. While the variable setting `3*` shows the state of the variables after executing code, `2*` displays the intermediate state after executing the first statement and before the second one. The numbers `1*-3*` are also shown within the derivation tree to indicate the semantics before and after executing a particular node.

only be a single vector of some values but multiple vectors, as a program contains multiple variables that undergo state changes. The variable setting `1*` contains the initial setting, which might be the training data. In this example, it consists of only three cases. The variable setting `2*` shows the state of the variables after executing the first line of the code, but before executing the second line. `2*` is the semantic output of the subtree to the left of the root node. The variable setting `3*` contains the state of the variable after executing both lines of code and is the semantics of the whole tree. Depending on the statement that is executed a variable can change its stored value in all, some or even no cases. Variable x is changed in all instances after executing the first statement, while variable a has only been changed in one of three cases.

1* to 3* are also shown within the GP tree to indicate at which point a specific variable state is established. A number to the left means the variable setting before and to the right after executing the node. Not every node has a number with a variable setting. To evaluate the semantic change when exchanging a subtree on the lower levels that cannot be executed on its own, the first parent node that can be executed is used.

Unlike the code example in Figure 7.1, most code as well as code evolved with the grammar design approach contains conditions and loops, which leads to code statements that are either not executed for every single training case or executed multiple times. If a statement is not executed, there is no information if the code would change the current state of variables and therefore is of no interest. If code is executed multiple times and all of the variable changes would be tracked, it would make semantics difficult to compare as the trace that is logged would be of different length. A decision was made to log the variables the first time a change in any variable appears when executing a statement, to avoid these issues.

An example of a code snippet containing a loop and an *if* condition is shown in Figure 7.2. Similar to the previous example the code, its corresponding derivation tree and the variable settings before and after executing different nodes in the tree are shown. Due to some statements being executed multiple times and the *if* condition, the example is more complex as the one in Figure 7.1. Note that the for loop already changes the variable i, so the first statement within the loop already has a different variable setting before it is executed. So, the node <code><statement> has variable setting 2* before it is executed and variable setting 4* after it is executed as semantics. The semantics after executing the for loop 5* is different from 4*, because only the first time <code><statement> is executed the variables are logged. The reasoning is the same for variable settings 3* and 4* being the same, although <code><statement> contains an *if* condition. The statements within the *if* condition are not executed in the first iteration, but variables have changed for its parent node. So these changes are saved for <code><statement>. Although the *if* condition is executed in every iteration, it does not change any variable. Therefore these executions are not logged. The first time the statements within the *if* statement are executed is when i becomes 4. This is the time the variable settings A* and B* are logged.

The definition of semantics for program synthesis presented in this section is used throughout this chapter. To use this semantic information within an oper-

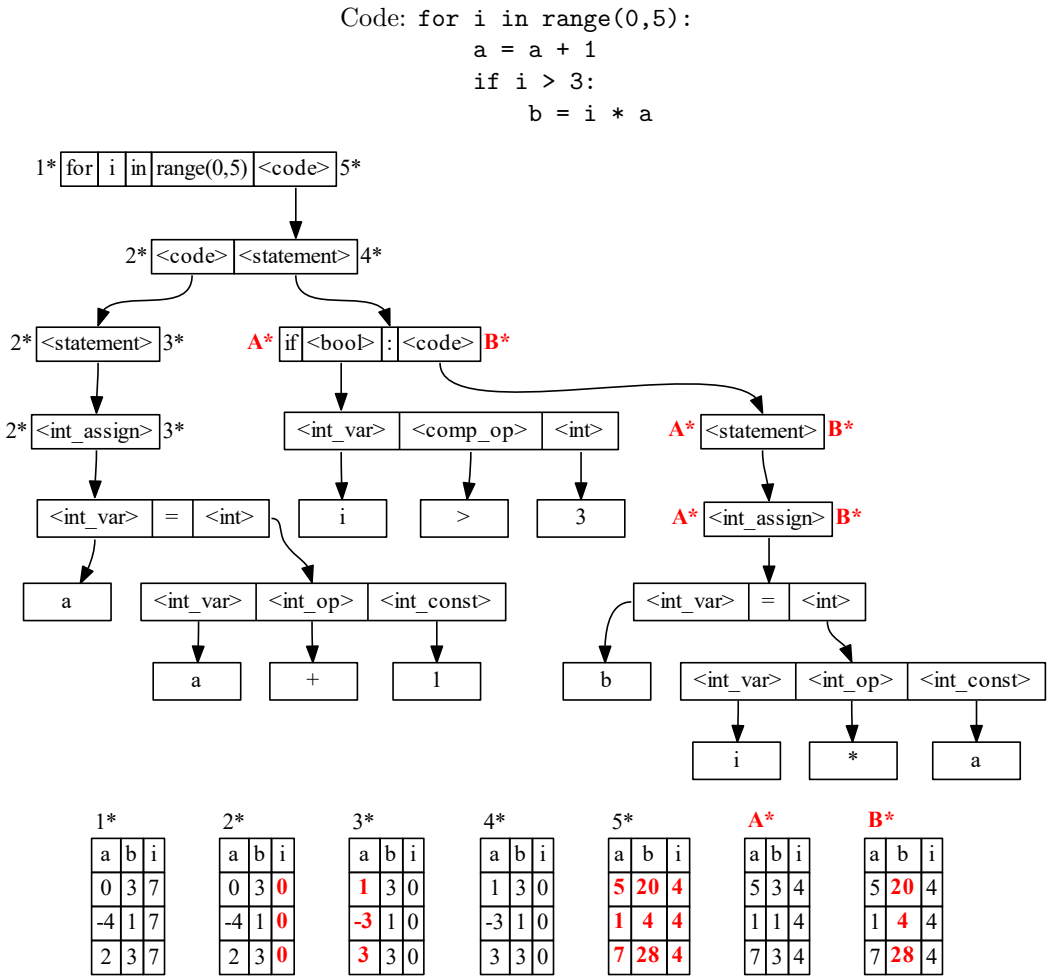


Figure 7.2: Program synthesis semantics example including a loop and an if condition. A sample of code containing a loop and an if condition is shown with its corresponding derivation tree and its trace for three different inputs. The state of the variables at different points of the derivation tree before executing the code is shown in variable setting `1*-5*` as well as `A*` and `B*`. Variable changes are only logged the first time a statement is executed and any variable changes.

Table 7.1: Similarity measures per variable

Variable type	Similarity measure
Boolean	Hamming distance
Integer	Sum of absolute differences
Float	Sum of absolute differences
String	Sum of Levenshtein distances
List of any type	Sum of Levenshtein distances

ator, a measure to compare the semantics of two programs is required. Different semantic measures have been tested and will be discussed in Section 7.3 and 7.4.

7.3 Semantic Crossover for Program Synthesis

The first semantic operator, Semantic Crossover for Program Synthesis (SCPS), presented is loosely based on Most Semantic Similarity-based Crossover (MSSC) by Nguyen et al. [34], which is summarised in Section 7.1.1. A semantic measure for program synthesis which is required to compare the similarity of two programs is described in Section 7.3.1. This measure will be used in SCPS. The operator SCPS is described in Section 7.3.2. The goal of this operator is to check if semantics in program synthesis can be used to improve performance similarly as in regression and boolean domain, to analyse its behaviour and to gain insights that can help to make SCPS more effective.

7.3.1 Semantic Measure

The semantics used for program synthesis is defined in Section 7.2. It shows that in contrast to regression and the boolean domain, which only use a single vector of one data type, real values and boolean respectively, program synthesis requires multiple vectors of different data types. Therefore, to compare the semantics of two programs, multiple similarity measures are needed. A suggestion for the similarity measures for each data type is shown in Table 7.1.

Most of the measures are well-known, like Hamming distance for boolean vectors or the sum of absolute differences for integer and float values, which is also used in MSSC. Strings can be compared with the Levenshtein distance. So for two vectors of strings, the sum over these distances is used. The last entry in Table 7.1

"List of any type", refers to semantics produced when using lists in a program. As two lists do not necessarily have the same number of values and list operations in a program can add, remove or change a value, the Levenshtein distances is an appropriate measure to capture the number of changes required to transform one list into the other. Similar to strings, the sum over all the distances is used.

Other distance measures can be used to compare the semantics of two programs. The ones in Table 7.1 are merely suggestions for the initial investigation of semantics in program synthesis and may be changed depending on the insights gained during the experiments.

It should be noted that it is not recommended to sum over the distances of different data types. The reason is that depending on the data type a different measure is used. While for boolean the Hamming distance can be between zero and the length of a vector, for integers and floats the sum of absolute differences can be between zero and infinity. Therefore, comparing two different distance measures seems not reasonable, which is why for each comparison in SCPS a single data type is chosen as explained in the next section.

7.3.2 Operator

The pseudocode in Algorithm 7.1 describes the proposed Semantic Crossover for Program Synthesis (SCPS). As with standard crossover, a crossover point from the first parent is selected. Then, instead of picking one random subtree from the second parent, which is of the same type as the selected node from the first parent, up to a maximum value of subtrees (*Max_Tries*) are chosen at random without repetition. *Max_Tries* is a parameter that can be set. If the second parent does not contain a subtree of the same node type as the selected one from parent one, no crossover is executed.

In the next step, the semantic differences between the subtree of the first parent and all the selected subtrees from the second parent are calculated. The calculation is performed in the following way. During the evaluation, the semantic information of every individual is saved in the form of an execution trace as explained in Section 7.2. The variable settings before and after the execution of each statement and therefore the corresponding subtree is saved as well. The variable setting before the execution of a subtree can be viewed as the *input* and the setting afterwards as the *output* of that code snippet. For each selected subtree

Algorithm 7.1 Semantic Crossover for Program Synthesis (SCPS)

```
select crossover point from first parent
select Max_Tries possible subtrees from second parent
if no subtrees of same type as crossover point available then
    no crossover
    return
end if
get semantics of every selected subtree from second parent
calculate semantic differences for every selected subtree per type
if differences then
    select random type
    select most semantically similar subtree based on selected type
else
    select random subtree for crossover from second parent
end if
crossover with selected subtree
```

Algorithm 7.2 Semantic similarity calculation for two subtrees

```
input1, output1 ← semantics of subtree from first parent
set variables to input1
output2 ← execute one subtree from second parent
calculate semantic distance between output1 and output2
```

from the second parent, the variables are set to *input* of the subtree of the first parent, followed by executing the subtree of the second parent and comparing the variable outputs to the *output* variable setting of the subtree from the first parent. A more concise description of this process is given in Algorithm 7.2 as pseudocode.

It should be noted that this is not a fitness evaluation as no fitness value is calculated, but a necessary process to find the semantic differences. If there is no difference for any subtree, one subtree is chosen randomly. If there is a semantic difference, a random data type is chosen, which shows a semantic difference. The reason why only one data type is chosen is that different data types use different distance measures and mixing them might result in unwanted behaviour. For all variables of the selected data type, the sum of semantic similarities is calculated with the corresponding similarity measure, shown in Table 7.1. The most semantic similar subtree that is not equal is chosen for crossover.

Parent code:

```
...
a = 7
b = a * 2 + 1 + x
n = b % 2 == 0
c = 5
...
```

Semantics before

x	a	b	c	n
2	7	1	5	False
1	7	1	3	False
3	7	1	5	False

Semantics after

x	a	b	c	n
2	7	17	5	False
1	7	16	3	True
3	7	18	5	True

Subtree code 1:

```
a = x * 8
n = a % 4 == 0
```

Semantics after code 1

x	a	b	c	n
2	**16**	**1**	5	**True**
1	**8**	**1**	3	True
3	**24**	**1**	5	True

Subtree code 2:

```
b = 16 + x
```

Semantics after code 2

x	a	b	c	n
2	7	**18**	5	False
1	7	**17**	3	**False**
3	7	**19**	5	**False**

Figure 7.3: Semantic Crossover for Program Synthesis example. Code of the parent, possible code to replace two lines marked in red within the parent and the corresponding semantics of all code pieces after being executed on the variable state of "Semantics before". Differences of the "Semantics after" of the parent code compared to code 1 and 2 are highlighted in bold red.

Figure 7.3 shows an example of a possible semantic crossover event. The two statements marked in red in the parent code have been selected for crossover and subtree code 1 and subtree code 2 selected as possible replacements. The semantics before and after the selected statements in the parent code are shown on the right-hand side and have been collected during the fitness evaluation. The semantics before is only required to be able to execute subtree code 1 and 2 on the same state of variables, which is done as described in Algorithm 7.2 which calculates semantics after code 1 and 2.

The semantics after the parent code is then used to compare to each semantics of the possible replacement subtrees and differences are highlighted in bold red in the figure. The variables that show a difference are then used for comparison. In the example, `a`, `b` and `n` show a difference in at least one semantics. `a` and `b` are of data type integer, while `n` is a boolean data type. In the semantics after code 1, `a` shows a difference of 27, `b` of 50 and `n` of 1, using hamming distance and sum of absolute differences for the corresponding data type, see Table 7.1. In the semantics after code 2, `a` does not show a difference, `b` has a difference of 3 and `n` of 2. This example shows why different data types are not just summed up for comparison, because the boolean data type can only have a maximum difference of 3 while the maximum difference in case of integer is infinite. Therefore, one data type is selected at random to compare the semantics of the subtrees. If integer is selected, subtree code 2 is most semantically similar as it only shows a difference of 3, while code 1 has a difference of 77 by summing over all differences from integer variables. If boolean is selected for the semantic similarity comparison, then code 1 is the most similar one as it has a difference of one compared to code 2, which has a difference of 2.

7.3.3 Experimental Setup

The goal of the experiments is to study SCPS and see if it can improve the performance of program synthesis tasks overall as well as if it can construct children that perform better than its parents. Further, semantic properties of SCPS are investigated as well as the semantic measure to find ways to improve SCPS. To this end, the G3P system with the grammar design pattern presented in Chapter 4 is used and a range of benchmark problems from the general program synthesis benchmark suite, described in Section 2.3.2, are selected. The problems chosen are of varying difficulty and require a mix of different data types, namely Checksum, Collatz Numbers, Compare String Lengths, Double Letters, Grade, Mirror Image, Number IO, Small Or Large, Sum of Squares, Super Anagrams and Vector Average. The experiments are executed with SCPS and standard crossover [54] for comparison.

The same parameter settings as in Chapter 4 are used and shown in Table 7.2. Lexicase selection is used, as it was shown to be more successful in program synthesis than tournament selection. Again three variables per data type will be

Table 7.2: Experimental parameter settings

Parameter	Setting
Runs	100
Generations	300 [1]
Population size	1000
Selection	Lexicase
Crossover probability	0.9
Mutation probability	0.05
Elite size	1
Node limit	250
Variables per type	3
Max execution time	1 second
Max_Tries	10

[1] 200 Generations for Number IO as set in [1]

available for storing temporary data and one second will be used as maximum execution time before a program will time out. The only additional parameter is Max_Tries for SCPS, which has been set to 10. Max_Tries is the number of subtrees that are selected from the second parent and compared to a subtree from the first parent. This number has been established in preliminary experiments. Up to 10 comparisons may seem high compared to standard crossover the chooses two subtrees and exchanges them without a single comparison, but as will be shown in Section 7.3.4, SCPS still has difficulty to find a subtree pair to exchange.

7.3.4 Results

In this section, the results of the experiments described in Section 7.3.3 are analysed. As mentioned, the overall goal of the experiments is to analyse and draw conclusions from the behaviour of a semantic crossover. Due to the additional computation that is required for the semantic crossover for program synthesis, an increased runtime is expected, but it has not been analysed, because the computational cost was added to collect the measurements which will be discussed in this Section. Additionally, the current implementation of semantic crossover has in no form been optimised.

Successful Runs and Fitness

Table 7.3 shows the number of times a run was able to find a correct solution to a problem for test and training, as well as the average test fitness of the best training solution, average training and test cases solved and the p-value from Wilcoxon rank-sum test on the average test fitness comparing SCPS with SC. When comparing SCPS to standard crossover, Table 7.3 shows that the correct solutions on test found on 100 runs is quite similar in most cases. In the case of Grade, the number of found solutions is smaller, but more solutions have been found on training, which means that although training was solved the solutions did not generalize. An issue also discussed in Chapter 5. Vector Average is the only problem on which SCPS does worse on all accounts, but the difference is not statistically significant in terms of test fitness of the best individual. Even after a thorough investigation, no explanation has been found, why this is the case. SCPS achieves more successful solutions on training on five of the problems with 33 more on Super Anagrams and over ten more on three problems. Even though the number of successful solutions is usually more important for program synthesis, the fitness gives an indication if a solution has gotten closer to solving all test cases. While the improvements on fitness with SCPS are on average higher than the decline, on two problems, the average test fitness of the best individuals decreases and the difference is statistically significant. On Double Letters and Sum of Squares, SCPS improves, and the difference is statistically significant.

Additionally, Figure 7.4 depicts the average best training fitness over 100 runs. The plots show that on average with SCPS better solutions are found in earlier generations, except on Vector Average. Although SCPS helps to improve solutions overall, the results are only slightly better than with standard crossover.

Parent Comparison

The goal of SCPS is to exchange similar subtrees, but not equivalent ones. Therefore, this change should be visible in the child by having a different semantics than its parent. McPhee et al. noticed in the boolean domain that more than 50% of standard crossover operations were not able to change the semantics [31] and Nguyen et al. reported that even though standard crossover was able to change semantics in the regression domain in 60%-80% of crossover operations, semantic crossover was often 20% higher [34]. Figure 7.5 shows the percentage of individ-

Table 7.3: Results on benchmark problems running G3P 100 times on each problem with SCPS. The table contains the number of successful runs on test and training data, the average test fitness and the average percentage of solved training and test cases of the best solution found during training with the improvement over standard crossover and the p-value from Wilcoxon rank-sum test on the average test fitness. The result is compared to standard crossover with the differences shown in brackets.

Problem Name	Test	Training	Avg Fitness (% Improv.)	Avg Solved Train.	Avg Solved Test	p-value
Checksum	0 (+0)	0 (+0)	35874.03 (+0.39%)	45.18% (+13.01)	21.28% (+9.08)	0.7573
Collatz Numbers	0 (+0)	0 (+0)	83351.78 (–0.38%)	1.98% (+0.31)	0.80% (–0.01)	0.8815
Compare String Lengths	4 (+2)	99 (+0)	128.05 (–15.25%)	99.99% (+0.00)	87.20% (–1.69)	0.6583
Double Letters	0 (+0)	0 (+0)	4361.29 (+6.36%)	24.43% (+0.80)	11.39% (–0.41)	**0.0377**
Grade	19 (–12)	95 (+14)	88.52 (+64.07%)	99.89% (+2.65)	98.23% (+2.83)	0.6407
Mirror Image	0 (+0)	63 (+12)	400.93 (–18.87%)	99.53% (+0.60)	59.91% (–6.37)	**0.0001**
Number IO	96 (+2)	99 (–1)	320.28 (+99.10%)	99.00% (–1.00)	98.98% (–0.57)	0.2778
Small Or Large	7 (+1)	72 (+13)	501.33 (+15.39%)	98.80% (+1.62)	89.83% (+2.16)	0.0993
Sum of Squares	9 (+6)	12 (+9)	140179.15 (+46.49%)	28.10% (+17.30)	24.76% (+15.90)	**0.0000**
Super Anagrams	0 (+0)	75 (+33)	273.51 (–2.92%)	99.87% (+0.22)	86.32% (–0.39)	**0.0464**
Vector Average	3 (–13)	3 (–14)	259913.73 (–14.55%)	28.51% (–6.18)	24.35% (–9.08)	0.0839

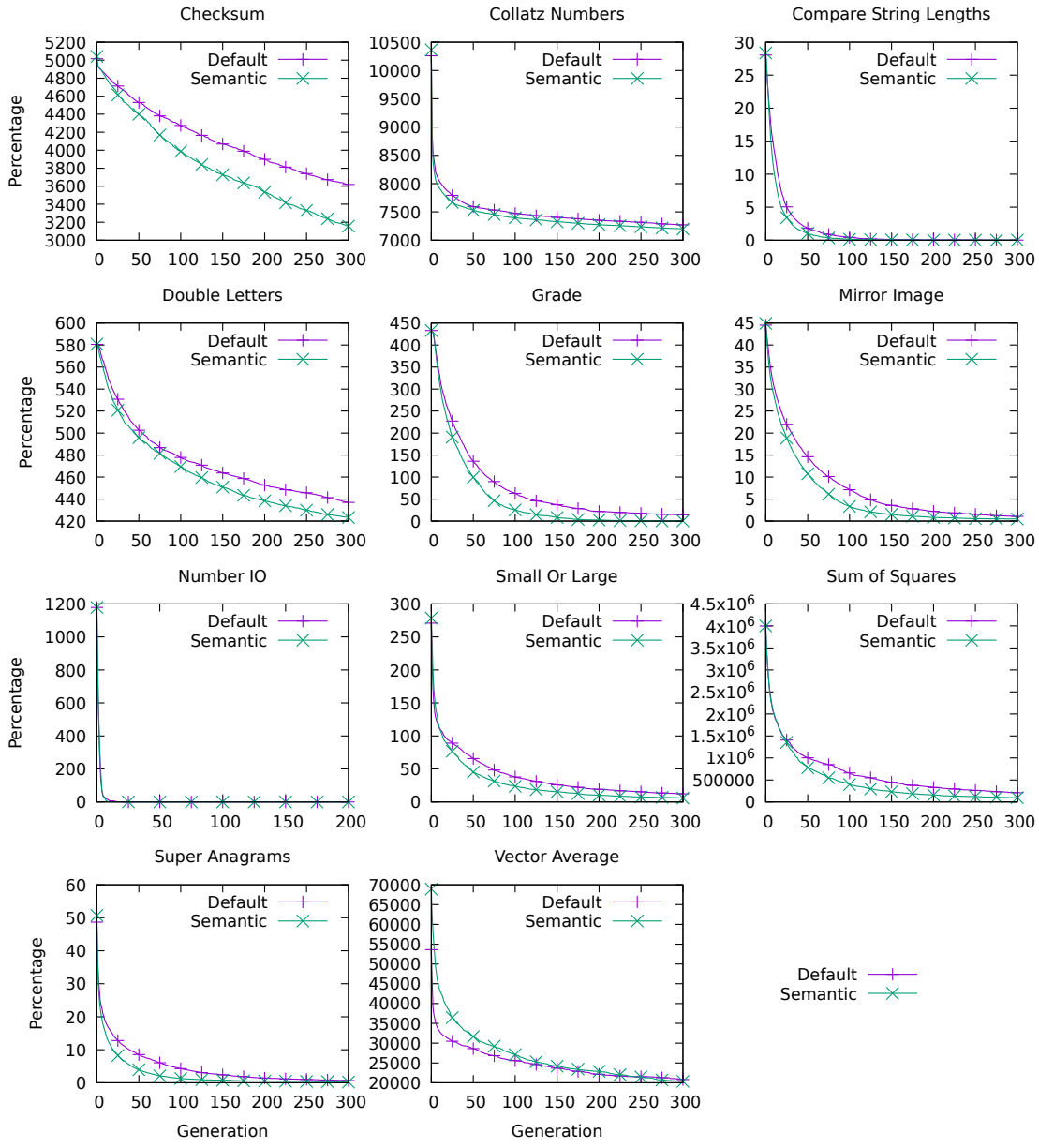


Figure 7.4: Average best training fitness over generations for SCPS ("Semantic") and standard crossover ("Default").

uals that are different from their rooted parent, with SCPS on top and standard crossover below. The rooted parent is the parent which removes a subtree to add the subtree from the second parent. Therefore the child and the parent have the same root node. For each problem, SCPS is able to create more children that are semantically different from their rooted parent. The percentage for Number IO is lower than the others because it is a rather simple problem and solved within the first few generations in most runs. It seems that when a run found a successful solution that it becomes more difficult to generate semantically different children. A similar case, but not as extreme, can be seen in Compare String Lengths, which has a decrease over generations. Similar to Number IO, Compare String Lengths is solved in almost all runs at least for training and the fitness is close to zero, as shown in Figure 7.4.

More interesting than just if a child is different than a parent, is if a child is better than its parents. Figure 7.6 shows the percentage of children that have better fitness than their rooted parent and better fitness than both parents. For many problems, like Checksum, Collatz Numbers, Double Letters, Sum of Squares and Vector Average, SCPS creates continuously more children that are better than their rooted and both parents than standard crossover. For other problems, SCPS still achieves higher percentages initially but drops after some generations. Sometimes the percentages become lower than with standard crossover. These drops only occur on problems that are solved quite frequently on training, like Double Letters, Grade, Mirror Image, Number IO, Small Or Large and Super Anagrams are all solved more than 50 times on training. The runs continue even after training is solved to collect these statistics. Because the runs continue, even though the run found a solution that solves training, it is not possible to produce a child that is better than a parent that already solves the problem. Also, SCPS often finds more solution on training. Therefore the percentage of children that are better than its parents may drop below the percentage achieved with standard crossover. In general, SCPS produces more children that are better than their parents.

Types Selected for Similarity Measurement

As described in Section 7.3, one data type of all available data types that shows a semantic difference is randomly chosen to measure semantic similarity. Figure 7.7

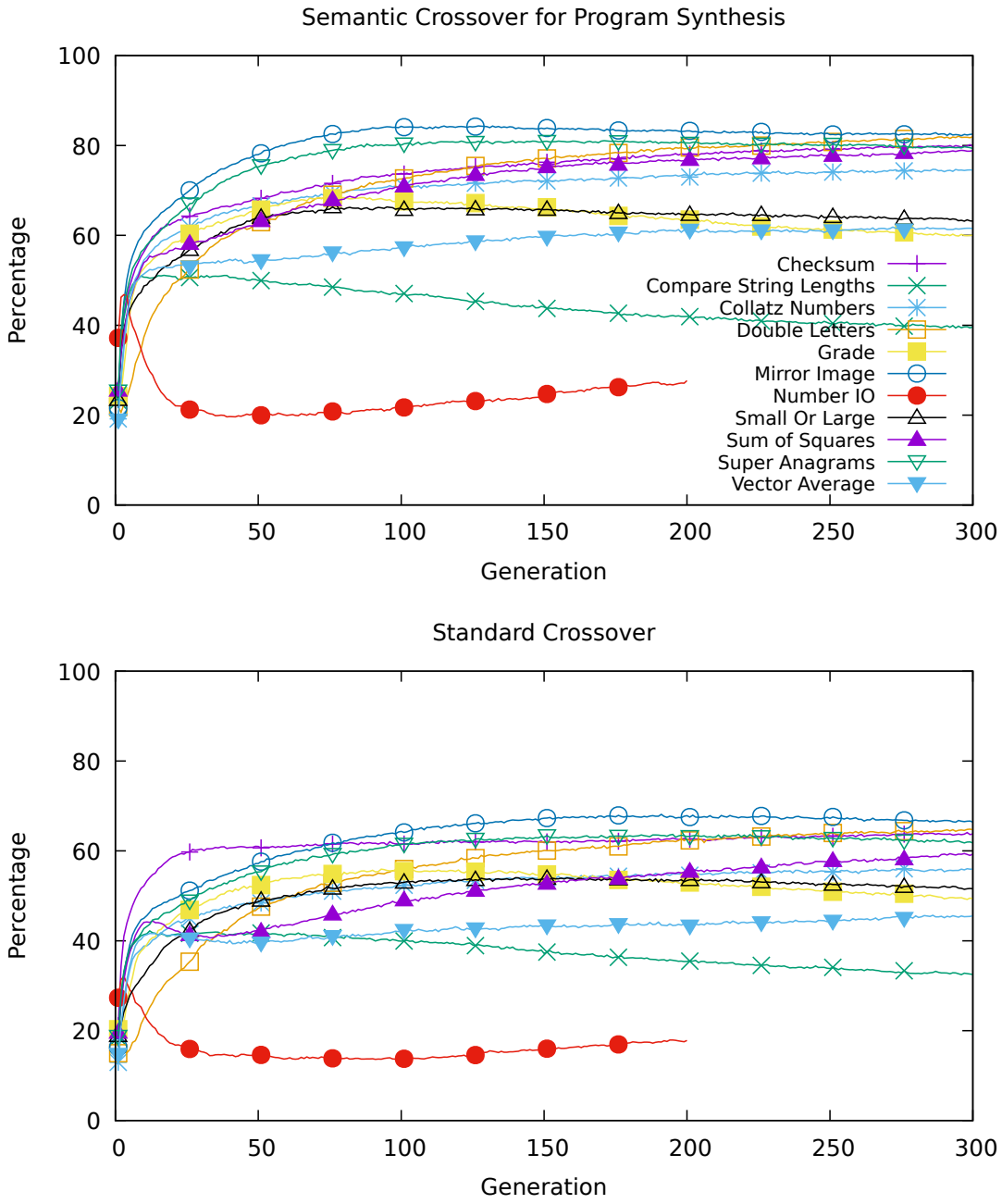


Figure 7.5: Percentage of children semantically different from their rooted parent. SCPS on top and standard crossover below.

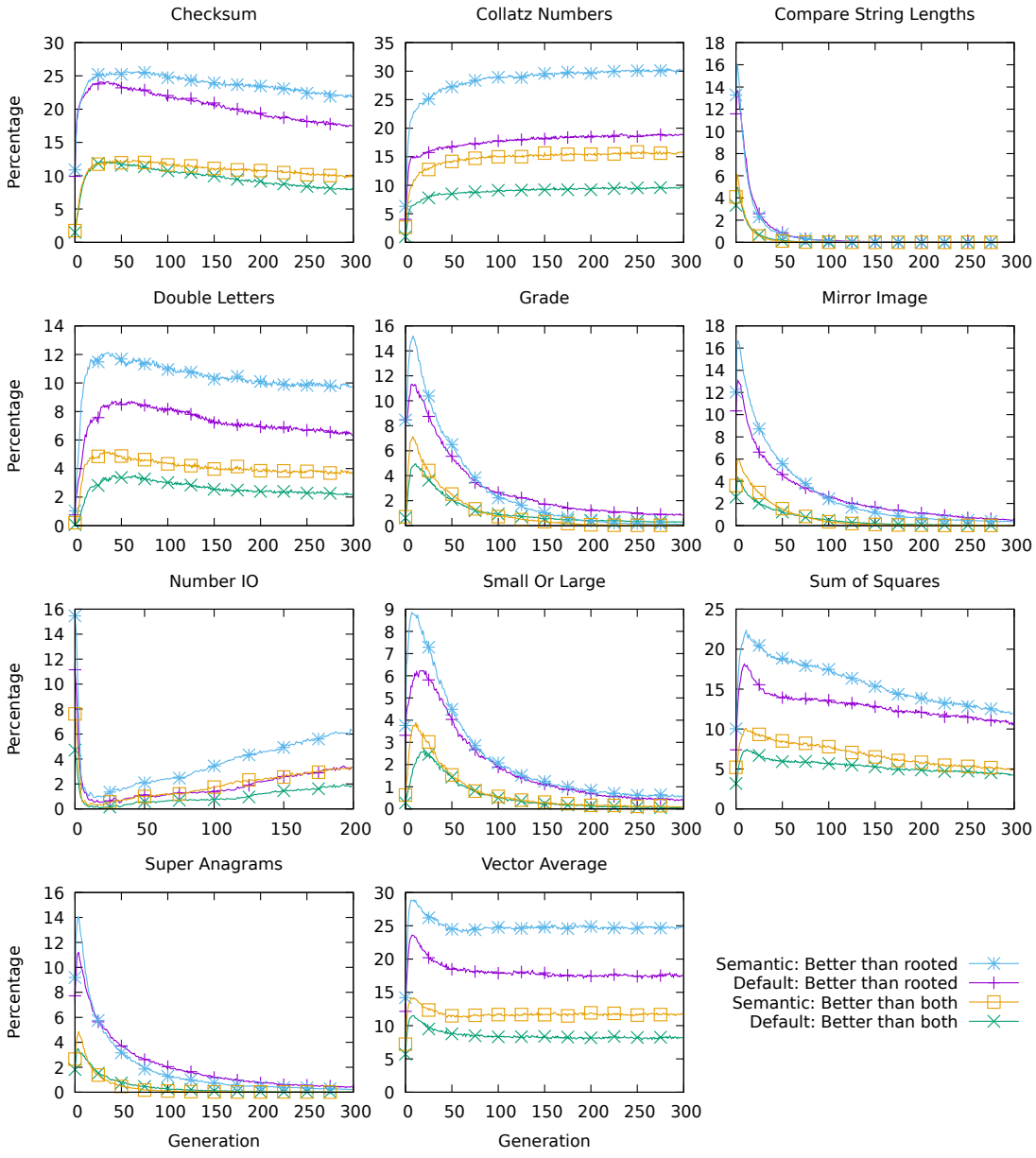


Figure 7.6: Percentage of children that are better than rooted or better than both parents for SCPS ("Semantic") and standard crossover ("Default").

shows the percentages of how often a particular type has been selected for the semantic similarity measure or if a random crossover or no crossover was executed. As mentioned before, no crossover can happen, if the second parent does not contain a subtree of the same type as the selected node from the first parent or no subtree of the same type applies the node limits set. Random crossover happens if no semantic difference can be found with any data type on the specified number of subtrees selected.

As can be seen in all cases, the data type that has been chosen most often for calculating the semantic similarity is the data type that is used as a return value for each problem and therefore has the most influence on the fitness. Although SCPS uses its semantic measure to choose a subtree in all cases more often than falling back to random crossover, the amount of random crossover is high on all problems, which happens because no semantic difference can be found with any data type. For many problems, 20-40% of random crossover is used. In the case of Number IO, SCPS falls back to random crossover 50% of the time.

The high percentage of random crossover indicates two things. First, that many crossover operations are not able to find subtrees that are semantically different, which means the individual produced will not be different from their parents. If the individual is not different from their parents, it can also not be better, which is why the percentage of children that are better than their parents is not higher, see Figure 7.6. Second, the semantic similarity measure used for SCPS is very detailed and might not be required to make such a complicated comparison. A relatively high number of subtrees that have been checked during the semantic crossover for program synthesis seem not to be able to create a semantically different individual. Adapting the crossover to using the first subtree that is semantically different instead of using the most semantically similar one might be sufficient and improve run time, which will be more similar to the original semantic similarity-based crossover proposed in [33]. Increasing the number of *Max_Tries* could also increase the number of times a semantic crossover finding a semantic difference, but that would increase run time.

7.3.5 Summary of SCPS

A semantic similarity-based crossover was adapted for the program synthesis domain. To this end, methods for semantic distance measure were proposed, which

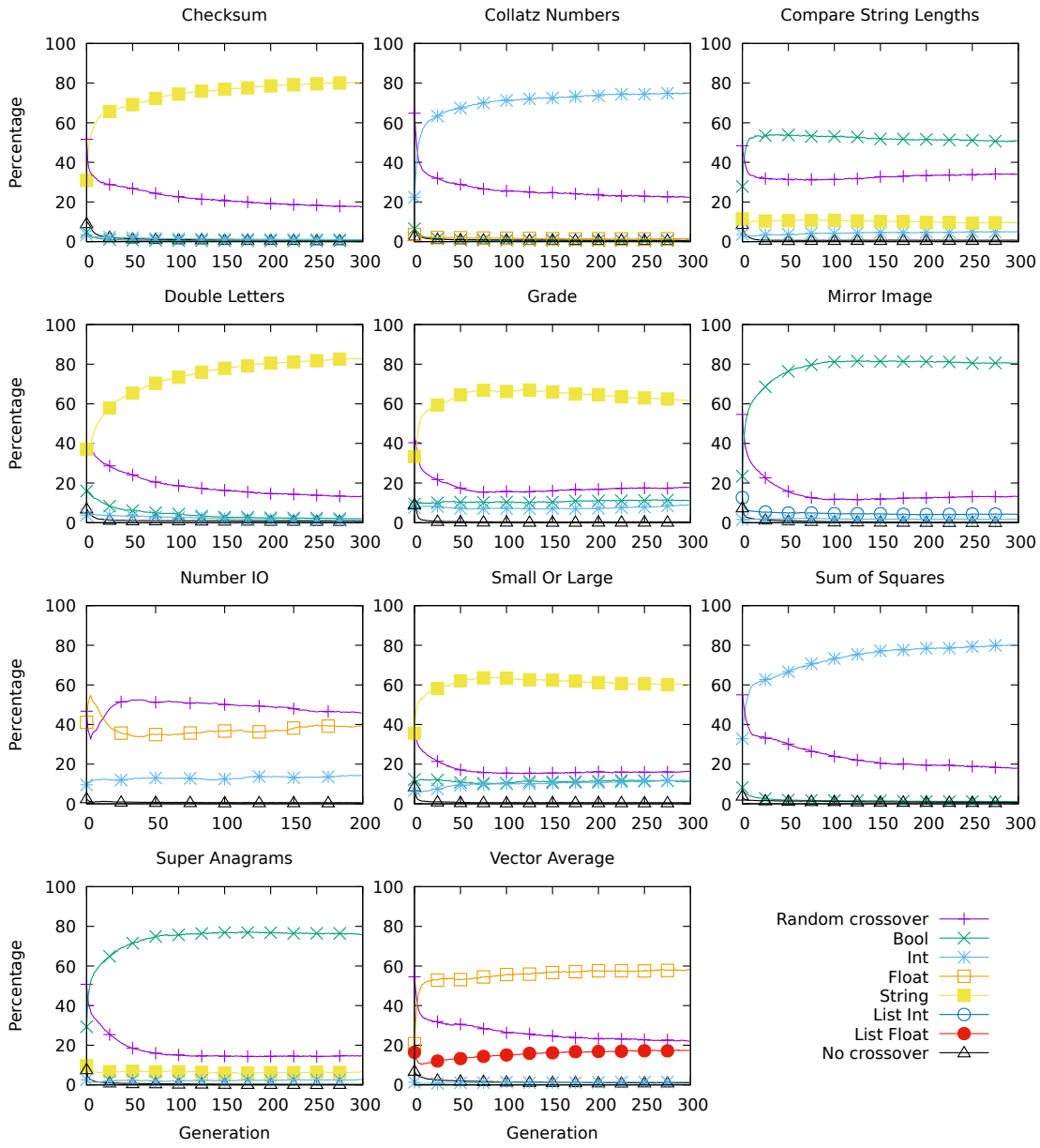


Figure 7.7: Percentage of crossover of a specific type with SCPS.

use the execution trace of a program, and the semantic crossover was applied to a suite of benchmark problems.

Semantic similarity crossover for program synthesis was able to produce more children that are semantically different from their parents as well as more children that are better than their rooted parent and both parents. Nevertheless, this did not lead to better overall performance. A reason might be that a high percentage of times the semantic crossover was still falling back to random crossover if it does not find any semantic difference on any selected subtree.

As mentioned before, the check for semantic similarity was complex and still a lot of times SCPS had to fall back to random crossover, because no semantically similar subtree could be found, even with up to 10 tries. A more straightforward check for semantic similarity like checking only for any semantic difference might be sufficient to improve performance and might reduce run time over the semantic similarity measure proposed for SCPS. Additionally, adapting the crossover to consider multiple different crossover points in the first parent instead of a single one might also lead to finding semantic differences more often. Improvements on SCPS will be studied in the next section.

7.4 Effective Semantic Operators for Program Synthesis

In this section, novel semantic operators are presented which use semantic information more effectively than the previously introduced operator SCPS. Using insights gained from the experiments with SCPS, shortcomings are addressed to create effective semantic operators. A semantic crossover is described in the next section, followed by a mutation operator that acts on a similar principle.

7.4.1 Effective Semantic Crossover for Program Synthesis

An adapted version of SCPS that fixes the issues discussed in Section 7.3.5 has been created and named Effective Semantic Crossover for Program Synthesis (ESCPS). Pseudocode for ESCPS is shown in Algorithm 7.3. The first small but important change is that similar to MSSC a pair of subtrees is selected. One subtree from each parent. The semantic information from the subtree of the first parent

Algorithm 7.3 Effective Semantic Crossover for Program Synthesis (ESCPS)

```
repeat
  select subtree from first parent
  select subtree of matching type from second parent
  calculate semantics of second subtree
  compare semantics for partial change
  if partial change found? then
    do crossover between subtrees
    return
  end if
until successful crossover or maximum tries
check all subtree pairs again for any difference
do crossover with the first pair that shows any difference
```

Algorithm 7.4 Calculate semantics for a subtree from the second parent

```
input1, output1 ← semantics of subtree from first parent
set variables to input1
output2 ← execute subtree from second parent
```

has already been collected during the fitness evaluation, so no further overhead is required for that part.

To be able to compare the semantics of two subtrees, both subtrees have to be executed on the same state of variables. It is important to note that this is not a fitness evaluation as no fitness value is calculated. The pseudocode that describes the process of establishing the semantics of the subtree from the second parent, which has also been used for SCPS, is shown in Algorithm 7.4. The only difference between Algorithm 7.4 and Algorithm 7.2 is that the last line has been removed. The semantic distance is not calculated within this algorithm anymore, as the semantics is used within ESCPS up to two times.

In the next step, the semantics of the two subtrees are compared. As explained in Section 7.3.5, the previous measure for similarity was rather complex and still was unable to find many subtrees that produced different semantics. Therefore, a more straightforward semantic measure that is easier to use and implement for various data types has been established that checks for a *partial change*. As the semantics of a subtree is a vector of values for each variable, the measure checks for every variable if there is at least one difference between the semantics from the subtrees in a single entry in the vector, but the vectors are not allowed to be completely different. Or to put it in other words, at least one entry has to

be different, and at least one entry has to be identical. This check provides the information that the subtrees are not equivalent but have some similarity.

If a partial change has been found, the two subtrees are used for crossover. No further steps need to take place. Therefore, the number of semantic comparisons can be smaller than the number of comparisons that need to take place with SCPS or even MSSC, which reduces the computational effort. Although in the worst-case scenario the number of semantic comparisons will be identical to SCPS and MSSC. As it is still possible to not find a partial change after a maximum number of tries and to avoid falling back to random crossover right away, an additional second semantic measure is used. The second semantic measure checks for *any change* in the semantics between two subtrees. The same subtrees are checked with the second measure and the first pair of subtrees that shows any difference is selected for crossover. The intention is to avoid falling back to random crossover and use the semantic information gathered in the previous loop. Additionally, the second semantic measure comes with little computational overhead, as all pairs of subtrees and the corresponding semantics have already been calculated.

Only if both semantic measures fail to find a partial or any change, crossover falls back to the default behaviour, which is selecting subtrees at random. So, one subtree pair that has been selected within the loop is used for crossover.

7.4.2 Effective Semantic Mutation for Program Synthesis

An Effective Semantic Mutation for Program Synthesis (ESMPS) operator has been created as well, because semantics can be used for mutation in a similar way to crossover. It works on the same principle as the ESCPS described above. Like conventional subtree mutation, a new random subtree is generated, but the semantics of the new subtree is evaluated to decide if it should be used. Again, a maximum number of tries can be set to do so and first ESMPS checks for a partial change as long as the maximum number of tries has not been exceeded. Afterwards, it falls back to check for any change, before it has no other choice than to fall back to random mutation. The pseudocode would be very similar to Algorithm 7.3, except the third line, would be replaced with “generating a random subtree”, which would be used in line four instead of the “second subtree”.

7.4.3 Experimental Setup

The goal of the experiments is to show that semantic operators are able to outperform conventional operators even in the program synthesis domain as well as that the semantic operators do not have to fall back to random crossover or mutation as it was often the case with SCPS. The same experimental setup, parameter settings and problems, is used as in Section 7.3.3, which shows the parameter settings in Table 7.2. No additional parameters are required for ESCPS and ESMPS except Max_Tries, which has already been added with SCPS. In contrary to SCPS, which selects up to Max_Tries subtrees and compares to all of the selected subtrees, ESCPS and ESMPS compare one subtree pair at a time, which therefore can decrease the number of comparisons required. The parameter settings are shown in Table 7.2.

7.4.4 Results

This section discusses the results of the experiments carried out with the effective semantic operators. The overall success rates, as well as semantic aspects of the operators, are discussed. The following results are mainly focused on crossover since it is the operator that is primarily used and therefore of higher interest. The plots for mutation are very similar and thus omitted from the chapter, but can be found in Appendix D.

Successful Runs and Fitness

The overall number of successful runs, the average test fitness of the best individuals, the average percentage of training and test cases solved for the semantic operators are shown in Table 7.4 as well as the improvements compared to conventional subtree operators. Additionally, a Wilcoxon rank sum test on the test fitness of the best training individuals was carried out to check for statistically significance.

The table shows that in almost all cases the semantic operators have improved the results on the benchmark problems. Sum of Squares gained the most successful runs due to the ESCPS and ESMPS. Some problems suffer from overfitting as is the case with the standard operators, but the increase of successful solutions found on training and improvements on average test fitness indicate that the semantic

Table 7.4: Results on benchmark problems running G3P 100 times on each problem with ESCPS and ESMPS. The table contains the number of successful runs on test and training data, the average test fitness and the average percentage of solved training and test cases of the best solution found during training with the improvement over standard crossover and the p-value from Wilcoxon rank-sum test on the average test fitness. The result is compared to standard genetic operators with the differences shown in brackets.

Problem Name	Test	Training	Avg Fitness (% Improv.)	Avg Solved Train.	Avg Solved Test	p-value
Checksum	0 (+0)	0 (+0)	33389.56 (+7.29%)	50.17% (+18.00)	26.78% (+14.58)	0.2168
Collatz Numbers	0 (+0)	0 (+0)	83548.64 (–0.61%)	1.96% (+0.29)	0.84% (+0.03)	0.2191
Compare String Lengths	4 (+2)	99 (+0)	93.73 (+15.64%)	99.99% (+0.00)	90.63% (+1.74)	0.1723
Double Letters	0 (+0)	0 (+0)	4362.08 (+6.34%)	24.04% (+0.41)	11.37% (–0.43)	**0.0334**
Grade	27 (–4)	79 (–2)	144.73 (+41.25%)	98.98% (+1.73)	97.29% (+1.89)	0.4648
Mirror Image	0 (+0)	67 (+16)	343.50 (–1.84%)	99.58% (+0.65)	65.65% (–0.62)	0.8221
Number IO	97 (+3)	100 (+0)	92.68 (+99.74%)	100.00% (+0.00)	99.84% (+0.29)	**0.0082**
Small Or Large	6 (+0)	66 (+7)	532.75 (+10.08%)	98.61% (+1.43)	89.30% (+1.62)	0.3614
Sum of Squares	13 (+10)	16 (+13)	157854.53 (+39.75%)	35.04% (+24.24)	30.78% (+21.92)	**0.0000**
Super Anagrams	0 (+0)	42 (+0)	264.44 (+0.50%)	99.67% (+0.01)	86.78% (+0.07)	0.9076
Vector Average	16 (+0)	16 (–1)	195073.05 (+15.38%)	38.82% (+4.13)	33.98% (+0.55)	**0.0154**

operators are beneficial for those problems as well. The average test fitness of the best training solution and percentage of solved test cases has not improved on all problems, but the amount it has decreased by is negligible compared to the improvements that were achieved in most cases. Only in case of Grade, the number of successful solutions found on training and test has slightly decreased, but the improvement on the fitness indicates that on average the solutions found were closer to solving the problem. Maybe with a bigger population size, a successful solution would have been found, see Chapter 5.

In Figure 7.8, notched box plots of the test fitness of the best individual are shown. The test fitness of the best individual in training of the runs with semantic operators (Semantic) are compared with conventional subtree operators (Default). In four cases, Double Letters, Number IO, Sum of Squares and Vector Average, the results with the semantic operators are significantly better than with subtree operators. The semantic operators were able to significantly improve on four of the problems tackled and had little effect on the others. So, in the worst-case scenario, some computational overhead is used with ESCPS and ESMPS, but the results do not become worse.

Semantic Measure Used

One of the shortcomings of SCPS was that standard crossover was used between 20-40% of the time, 50% for Number IO. ESCPS tries to address this issue. Figure 7.9 shows the semantic measure used for comparison. “Partial change” is the default semantic measure used, as explained in Section 7.4.1. If no subtree with a partial change is found, “Any change” will be accepted. As a last resort, standard crossover will be used. It should be noted that it is possible, that no crossover happens if no subtrees of the same type can be found that applies to the node limit but this only happens in rare cases.

For all problems “Partial change” is used most of the time, in many cases close to 100% of the time. This shows that this semantic measure is able to use the semantic information of subtrees more often than in the case of SCPS in Section 7.3.4. The second semantic measure “Any change” is rarely used and only during the initial generations of GP because “Any change” is solely used if “Partial change” fails and “Any change” is a slightly more general measure than “Partial change”. Even with this additional semantic measure, random crossover might be

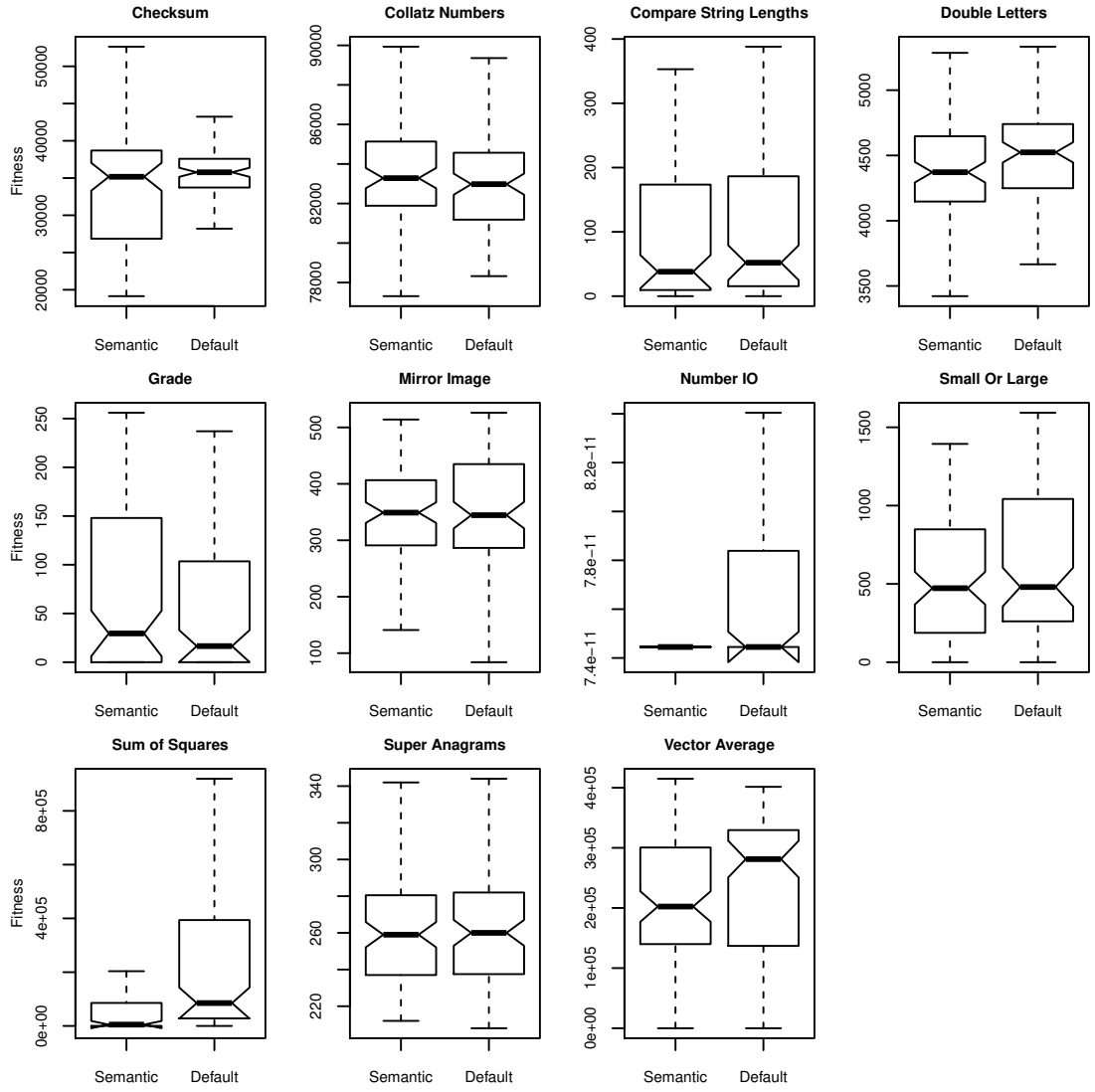


Figure 7.8: Notched box plots of the test ftness of the best individual during training comparing the semantic operators ("Semantic") to the syntactical subtree operators ("Default"). Outliers have been omitted for visibility.

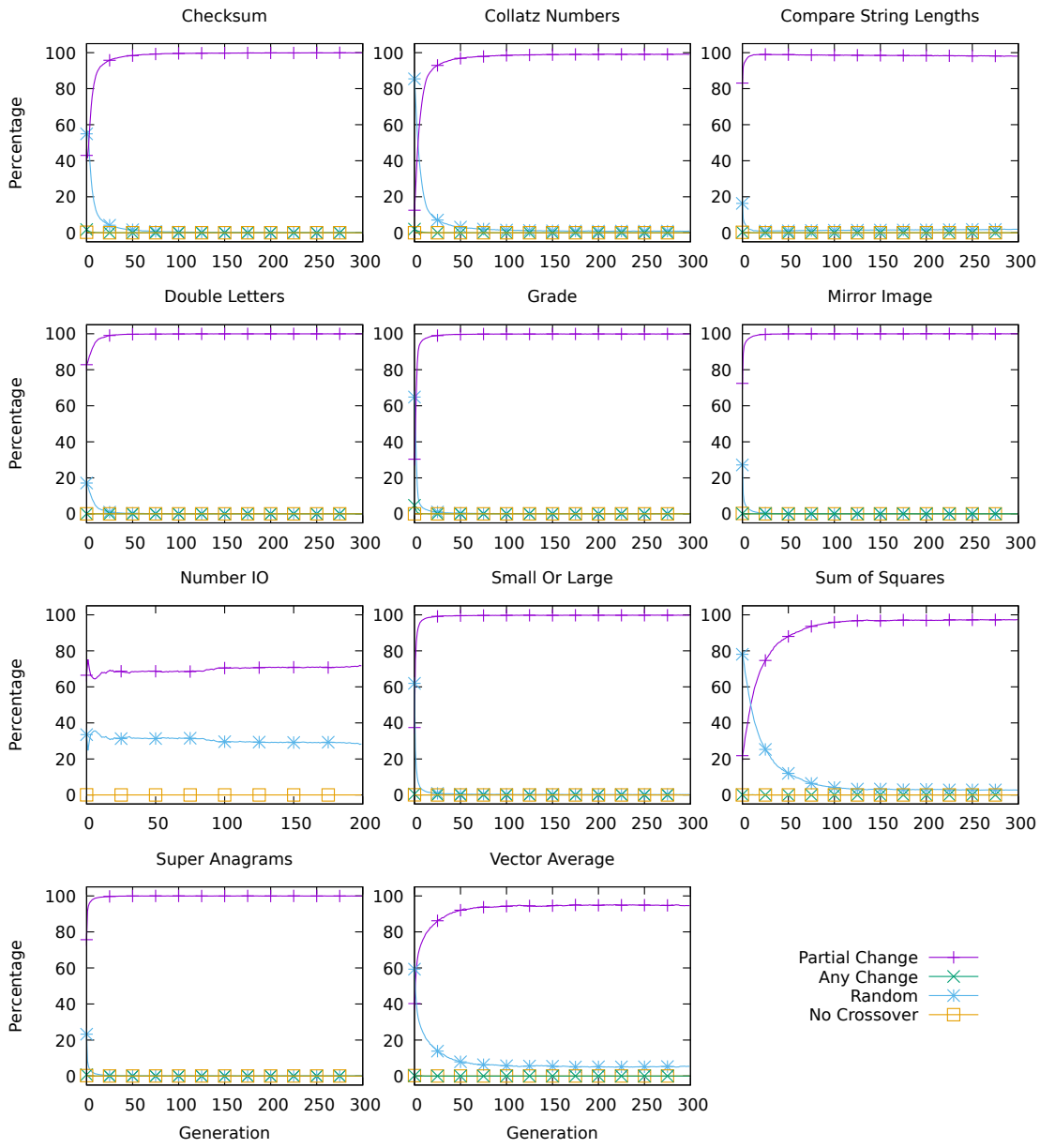


Figure 7.9: Percentage of semantic measure used during crossover over generations. "Partial change" is the semantic measure that is used first. If no subtree pair for crossover is found, "Any change" is used before falling back to "Random" crossover. 'No crossover' indicates that no crossover has taken place.

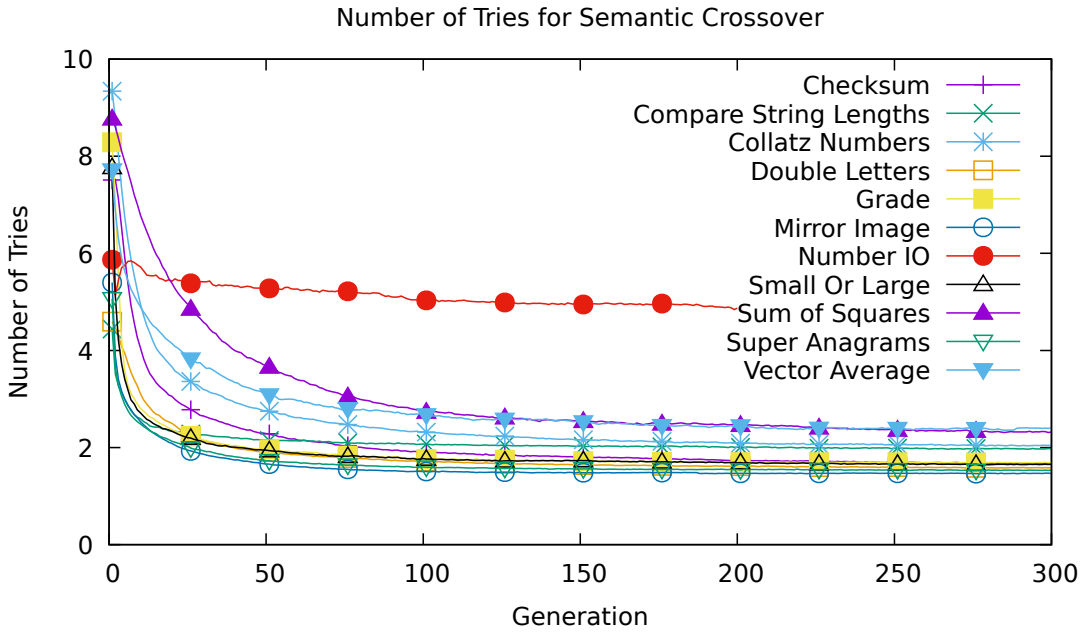


Figure 7.10: Average number of tries subtrees were selected for semantic comparisons until a subtree pair was used for crossover.

used more often, during the first few generations, but declines quickly. The only exception is Number IO, which continually uses standard crossover 30% of the time. This means that ESCPS is not able to find subtrees that are semantically different in 30% of the crossover events which is still a decrease of 20 percent point compared to SCPS. SCPS had to fall back to standard crossover about 50% of the time. In Section 7.4.4 it will be shown that even standard crossover has the least amount of semantically different children on Number IO.

Semantic Comparisons

Semantic comparisons are computationally expensive and a drawback of semantic operators. Figure 7.10 shows the average number of semantic comparisons that were required until a pair of subtrees were selected for crossover for all problems. If the second semantic "Any change" was used, the number of comparisons was already at the maximum of 10.

Figure 7.9 showed that in the initial generations it is more difficult to find a pair of subtrees with semantic differences, which explains why initially the number of comparisons is high in Figure 7.10, but it quickly declines for all problems and

stabilizes around 2 to 3. This shows that the number of comparisons is not even close to the maximum except in the initial generations and that the computational overhead of the semantic operator is on average low and less than for SCPS. The only exception again is Number IO, which usually would not be running longer than a few generations anyway, because solutions to that problem are found within the first few generations.

Parent Comparison

When the semantic operator is used and does not fall back to the conventional operator, the subtrees that are exchanged have different semantics. But this does not mean that the overall semantics of the whole individual changes. Figure 7.11 depicts the percentage of children that have semantics different compared to their rooted parent for all problems, showing ESCPS on top and standard crossover below. The rooted parent is the one a subtree is removed from, and the child shares the same root node with.

The percentage of children that are semantically different from their rooted parent is higher than with standard crossover as well as SCPS, see Figure 7.5. It is not 100%, because not every semantic operation on a subtree automatically leads to a change in the overall semantics, but it gets close to 100% for some problems. In some cases, the percentage declines slightly over generations. This only affects problems for which solutions were found that solve training. It seems that problems are more affected the more often they are solved, which might explain why Number IO has such a low percentage of semantically different children.

Changing the semantics of an individual is essential to keep semantic diversity high but does not automatically lead to better solutions. Figure 7.12 shows the percentages of children that are better than their rooted parent and both parents for crossover. For all problems, ESCPS achieves a higher percentage of children that are better than their parents than standard crossover.

It is expected that lines decrease over time as runs will have solved the problem and continue for the purpose of this analysis, even though the run will not be able to create better individuals. This is even the case when a solution only solves all training but not test cases. The same effect could be seen with SCPS in Figure 7.6. An extreme case is Compare String Lengths, where 99 runs have been able to solve the training data. The percentage of children that are better than their parents

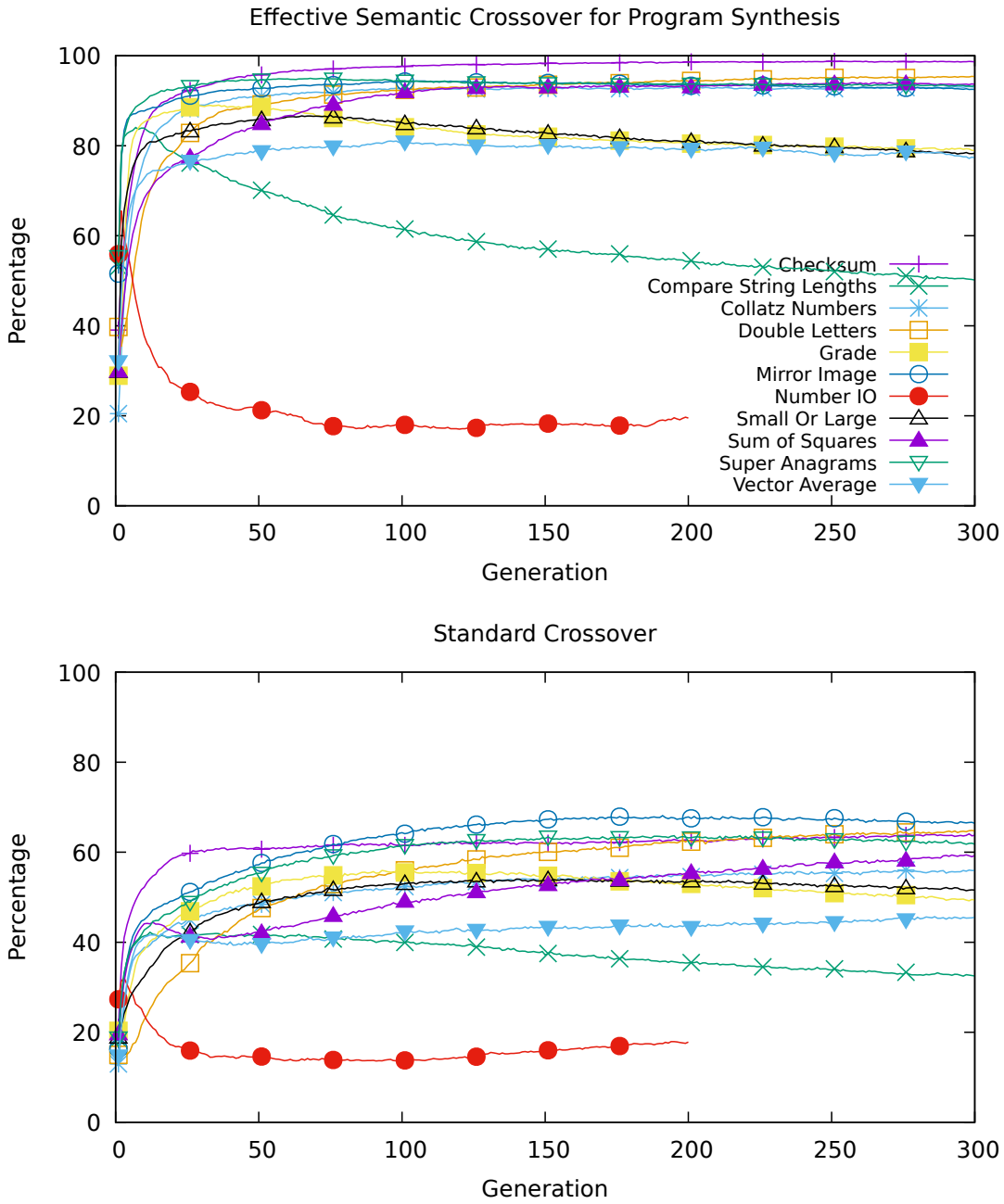


Figure 7.11: Percentage of children semantically different from their rooted parent with ESCPS (top) and standard crossover (bottom).

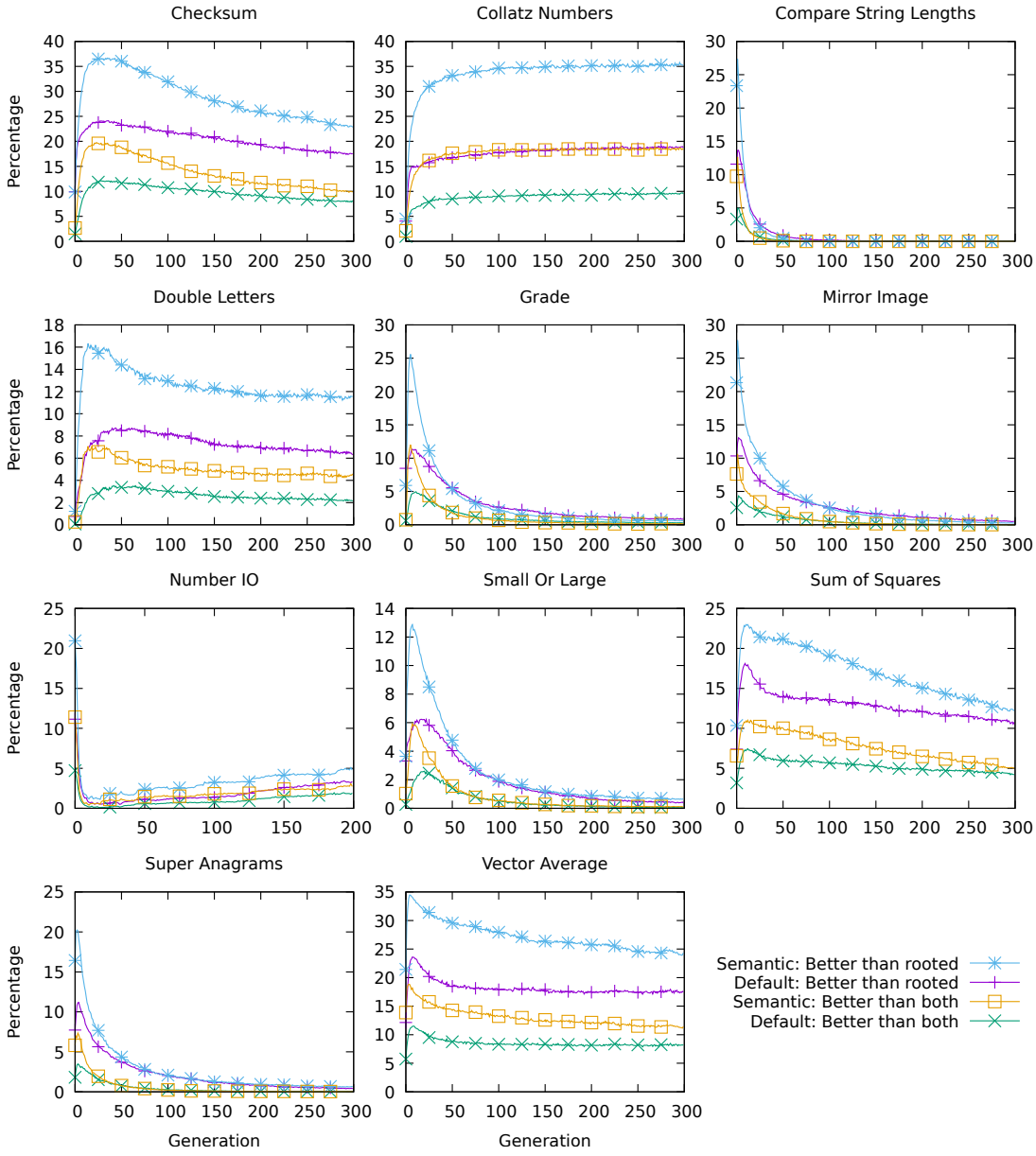


Figure 7.12: Percentage of children that are better than their rooted parent and both parents over generations created with crossover. “Semantic” for ESCPS and “Default” for standard crossover.

rapidly decreases and gets close to zero. In that case, even standard crossover achieves a higher percentage than semantic crossover, but only because more runs with ESCPS have already been solved, at least in training. A similar effect can be seen for Grade, Mirror Image, Small Or Large and Super Anagrams.

7.4.5 Summary of Effective Semantic Operators for Program Synthesis

Novel and effective semantic operators for program synthesis, ESCPS and ESMPS, have been introduced. These operators are adaptations of the previously studied Semantic Crossover for Program Synthesis (SCPS) and improve it by addressing its shortcomings. ESCPS and ESMPS are able to effectively use the semantic information available almost all of the time in contrary to SCPS by using a simpler semantic measure and selecting pairs of subtrees instead of comparing a single subtree to multiple others. These effective semantic operators were able to produce more children that were improvements over their parents as well as achieve statistically significantly better results than conventional subtree operators.

The results show that effective semantic operators for program synthesis can be created, but it would be helpful to know on which kinds of problems it is more likely to improve performance with semantic information. Additionally, it should be tested, if it is possible to create a semantic measure that can estimate the similarity between two subtrees more precisely than the partial change or any change measure used so far and still be effective, as semantic locality is of importance to improve performance.

7.5 Summary

Semantics and how it can be used in the domain of program synthesis has been studied in this chapter. A definition of semantics in program synthesis has been given in Section 7.2. Based on that definition multiple semantic operators have been introduced. SCPS is the first approach of having a semantic operator for program synthesis, and it is loosely based on MSSC, a semantic crossover from the regression domain. SCPS improved results on a set of benchmark problems, but in most cases not significantly and some result got worse. Nevertheless, SCPS provided the first insights about semantics in program synthesis. SCPS could

produce more semantically different children than standard crossover, and more of the children produced outperformed their parents.

Using the insights gained from SCPS, an improved crossover was created and named ESCPS, as well as a mutation operators ESMPS. Experiments conducted with both semantic operators either improved results on a benchmark problem or it showed not much of a difference. These experiments proved that improvements with semantic operators in program synthesis are possible. The definition of semantics in program synthesis, the created operators and the experiments conducted in this chapter builds a good foundation for further research in the area of semantics in program synthesis.

This chapter concludes the extended experimental research conducted with the grammar design approach. Part IV consists of the last chapter about the conclusion and future work as well as the appendices with grammars used and additional plots from experiments and finally the bibliography.

Part IV

Fin.

Chapter 8

Conclusion & Future Work

After having explored a grammar design approach to tackle program synthesis and studied semantics in the program synthesis domain, this chapter discusses the conclusions. Section 8.1 summarizes the thesis, its contributions and insights. Possible limitations are outlined in Section 8.3. Finally, future work is discussed in Section 8.4.

8.1 Thesis Summary

The overall goal of the thesis was to investigate a flexible grammar-based approach to tackle arbitrary program synthesis problems with a single reusable grammar as well as defining and exploiting semantics in the program synthesis domain to improve performance. For this purpose, four research questions have been stated in Chapter 1, which have been addressed throughout this thesis.

How can Grammar Guided-Genetic Programming (G3P) be utilized to tackle program synthesis?

Chapter 4 introduced a grammar design approach that is flexible and has been proven to be capable of tackling a range of program synthesis problems successfully. Results confirmed that performance similar to the state of the art is achievable, while no tailoring of the grammars as in previous approaches is required. Further studies in Chapter 5 and Chapter 6 investigated in two directions. Chapter 5 explored the computational effort required to solve program synthesis problems as well as showed some underlying issues with overfitting. Additionally, suggestions for improving

performance and counteracting overfitting were given. Chapter 6 extended the grammar design approach showcasing its flexibility as well as improving results on a range of problems that had not been solved before as well as solving one problem the state of the art system PushGP has not yet solved.

Do different derivation tree structures defined with grammars influence the search performance?

The first study undertaken, in Chapter 3, explored the influence of different derivation tree structures produced by grammars on the search performance. Five different grammars for sorting were created each of which produced different derivation trees but the same language. A significant difference in performance could only be found with Koza-style crossover and not with a crossover that chooses from all nodes in a tree uniform randomly. A similar experiment was carried out in Chapter 4 on program synthesis problems with the grammar design approach. This second experiment showed little performance differences using grammars producing different derivation tree structures in the program synthesis domain.

How to define semantics and semantic measures in GP for program synthesis?

Semantics for program synthesis was thoroughly investigated in Chapter 7. A definition of semantics in the domain of program synthesis has been stated, which allowed the exploitation of semantic information in that domain. Multiple semantic measures have been suggested and analysed with the grammar design approach.

How can semantic information be exploited in operators to improve performance?

Based on the definition of semantics in program synthesis in Chapter 7, search operators have been introduced. While the first semantic crossover operator in Section 7.3 displayed limited improvements to standard operators, the insights gained from the experiments were used to propose an enhanced version of the initial crossover as well as a semantic mutation operator. Section 7.4 explains the changes to the initial semantic crossover in detail and shows that semantic operators can outperform standard operators in the program synthesis domain.

8.2 Contributions

The main contributions of this thesis are listed below. Additionally, many of these contributions have been published in form of papers, which are listed on page xx.

Literature review

An overview of the most relevant topics, genetic programming, program synthesis and semantics is given in Chapter 2. The chapter reviews work from the program synthesis domain, outlines relevant methods to tackle program synthesis with, especially in the area of GP, and surveys work on the topic of semantics.

Study on derivation trees in G3P

Different derivation tree structures produced by grammars have been studied on the problem of sorting networks in Chapter 3 as well as program synthesis in Chapter 4, which showed that the form of recursive rules in grammars has little influence on the performance of G3P.

Grammar design approach

A novel approach to tackle arbitrary program synthesis problems with a single set of reusable grammars was introduced in Chapter 4, which can compete with the state of the art system PushGP and also produce code in a programming language used by practitioners. This approach was studied in depth in Part III of the thesis. Chapter 5 studied the computational effort required to solve program synthesis problems as well as analysed overfitting. Chapter 6 showed that G3P with a similar function set to PushGP, can solve more problems from a benchmark suite. It was also shown that the grammars can easily be extended if required. Finally, Chapter 7 investigated the use of semantics in the program synthesis domain with the grammar design approach.

Grammars for general purpose programs in Python

The grammars produced for the grammar design approach to evolve general purpose programs in Python have been made available in Appendix B and extended grammars in Appendix C, as well as online for public use [37].

Insights into the general program synthesis benchmark suite

Experiments conducted in the area of program synthesis use the prob-

lems available in the general program synthesis benchmark suite, see Section 2.3.2. All experiments carried out on this set of problems have given further insight in the benchmark suite, especially because the grammar design approach was the second method tested on these problems. Certain shortcomings of the benchmark problems and suggestions to improve them have been made in Chapter 5, e.g. adapting the computational effort depending on the difficulty of the problem as well as adjusting the dataset sizes to counter overfitting.

Definition of semantics in program synthesis

A definition and detailed description of semantics in program synthesis has been given in Chapter 7. Based on the definition further research can be conducted, even independently of the grammar design approach.

Novel semantic operators for program synthesis

Novel semantic operators for program synthesis have been introduced and improved in Chapter 7 based on the definition of semantics in program synthesis. Chapter 7 proved that semantics can be used in program synthesis to improve performance.

8.2.1 Technical contributions

All the implementation done during the completion of this thesis has been made available online on GitHub [37], which includes plugins for a heuristic and evolutionary algorithms framework called HeuristicLab [38], which are publicly available and open source. These plugins include all the code necessary to rerun any experiment conducted for this thesis, provide support for grammar-based problems to HeuristicLab, include the grammars and an automatic grammar combiner, lexicase selection as well as all the problems from the general program synthesis benchmark suite, discussed in detail in Section 2.3.2.

Parts of the implementation have also been integrated into PonyGE2 [39] as a showcase that other systems can quickly adopt the grammar design approach.

8.3 Limitations

This thesis covers a very broad and computationally expensive problem domain and introduces semantics in the area of program synthesis which leads to the consideration of certain limitations.

First, a derivation tree-based G3P, similar to CFG-GP [17] was used throughout the thesis. It has not been tested if the grammar design approach behaves similarly with a linearised G3P system like grammatical evolution.

The experiments conducted in this thesis on program synthesis only used problems from the general program synthesis benchmark suite [103] due to the lack of better benchmark problems in GP [40, 41, 42]. On the one hand, the benefit is that the results can be compared to other systems using the same benchmarks. On the other hand, the risk of missing an important problem instance that behaves differently exists.

The grammar design approach has only been compared to PushGP [95] in this thesis. Comparisons of program synthesis systems have been done by Thomas Helmuth in his PhD thesis [151] and Pantridge et al. [136]. Helmuth discussed the limitations of a range of program synthesis systems compared to GP approaches. Pantridge et al. showed the difficulty of comparing program synthesis systems. Nevertheless, a comparison was conducted including results from the grammar design approach [128]. The grammar design approach and PushGP were outperforming other methods used in the comparison. When comparing the grammar design approach to PushGP, the results from [1] have been used, although more recent results are available [152], because the more recent results also include additional optimizations not yet used in the grammar design approach.

Although the whole Chapter 7 is dedicated to investigating semantics in the program synthesis domain, many avenues of exploration have not yet been covered. Therefore many possibilities for future work exist.

Finally, no parameter optimization has taken place before the experiments. Most of the parameter settings have been taken from literature, the benchmark suite description [103] or have been established through preliminary experiments. The reasons why no parameter optimization has been done are partially due to the computational expensive nature of the problem domain and because confirming the successful application of and studying the grammar design approach was of

higher importance. The grammar design approach might achieve better results through parameter optimization.

8.4 Future Work

The work presented in this thesis opens up many avenues for further research. The flexible grammar design approach presented was only used on a benchmark suite of general program synthesis problems. Even though this suite consists of programming problems tackled by students and provides a way to compare different program synthesis systems, the problems have limited real-world application. Testing the approach on real-world problems might not only provide insights for research but would also be of interest to practitioners. Many applications in the area of software engineering are tackled with evolutionary algorithms already [98]. The grammar design approach might be able to help with new applications or improve on existing ones.

As shown in Chapter 6, grammars can easily be extended, but this increases the search space as well. The extension of the grammars lead to solving previously unsolved problems, but the reduction of success rates of others. Automatically adjusting the grammars and the functions used in the grammars could solve this problem. A technique named multi-level grammars by Saber et al. [130] allows using grammars of different sizes. Using such a method could allow introducing grammars with a larger function set if the problem cannot be solved with a smaller one. A possibility would also be to reduce the size of the grammars over time by removing functionality.

No parameter optimization has yet been undertaken as stated in Section 8.3. A parameter optimization could be a simple way to increase the success rates of the grammar design approach further. Rather than doing an exhaustive search of all possible parameter combinations, which would be very time-consuming in the program synthesis domain, systems to automatically configure optimization algorithms already exist, e.g. iterated racing for automatic algorithm configuration (irace) [153] and Sequential Parameter Optimization Toolbox (SPOT) [154]. Additionally, no post-processing was used so far, which can improve success rates as shown with PushGP [135]. It is worth investigating if such improvements also occur with the grammar design approach.

As already discussed in Chapter 4 and in more detail in Chapter 5, overfitting is an issue in program synthesis with GP, not just with the grammar design approach, but also PushGP is facing similar issues [1]. While some work has already been done, see Chapter 5 and [135], it is far from solved. Investigating issues causing overfitting and how to counteract it, would be of importance to the program synthesis community.

Semantics has helped improve performance in many problem domains and Chapter 7 showed that even in the program synthesis domain semantics can be exploited to achieve better results than with traditional operators. While the work in this thesis gave interesting insights on how to use semantics in program synthesis, there are still many avenues of research to explore. One is semantic initialization because it was shown that semantic diversity is more important than syntactical diversity [16]. Also, fine-tuning of the semantic operators, especially the semantic measures, presented in Chapter 7 to further promote semantic locality could help in solving more problems. Another aspect of semantics in program synthesis that should be investigated is the amount of semantic information that has to be saved to gain performance improvements. While the semantic information of a program can be collected during the evaluation, depending on the number of variables used and the number of assignments of values to variables, the more information has to be stored, which might end up to be a memory issue. To reduce the memory usage and maybe even increase runtime, only the semantics of a certain percentage of training cases may be stored.

Appendix A

General Program Synthesis Benchmark Suite Problem Description & Fitness Functions

A description of every problem of the general program synthesis benchmark suite is given in Section A.1 as well as its fitness function is listed in Section A.2.

A.1 Problem Description

Description of the general program synthesis benchmark suite [103] problems with returning instead of printing values:

Checksum
: Return a single character that is the sum of all integer values of the characters of the input string modulo 64 plus 32 converted to ASCII.

Collatz Numbers
: Return the number of terms in the Collatz conjecture sequence given an integer.

Compare String Lengths
: Return a boolean value indicating if three given strings fulfill the following condition: $length(string1) < length(string2) < length(string3)$

Count Odds
: Return the number of odd values in a list of integers.

Digits

Return the single digits of a number starting with the least significant digit. The most significant digit has to include the negative sign if the given number is negative.

Double Letters

Return the given string, but doubling every letter character and tripling every exclamation point.

Even Squares

Return a list of positive even squares that are smaller than a given number.

For Loop Index

Return a list of integer given a start, end and step size value.

Grade

Return a letter grade (A, B, C, D, or F) given five integers. The first four integers represent the thresholds between two grades and the last value is the grade as a numeric value.

Last Index of Zero

Return an integer representing the last occurrence of 0 in a vector of integers.

Median

Return the median value given 3 integer values.

Mirror Image

Return a boolean value indicating if two given lists of integers are the reverse of each other.

Negative To Zero

Return a list of integers that is the same as the given list of integers, except that negative values have been replaced with zero.

Number IO

Return a float that is the sum of a given integer and float value.

Pig Latin

Return a string that is the given input string translated to pig Latin. Every

word starting with a vowel has "ay" added at the end. Every other word has its first letter added at the and followed by "ay".

Replace Space with Newline

Return a string that has space characters replaced with newline characters. Additionally, return an integer representing the number of non-whitespace characters.

Scrabble Score

Return the Scrabble score for a given string of visible ASCII characters. Each character has a corresponding value in Scrabble.

Small Or Large

Return "small", "large" or a string of length zero depending if a given integer is smaller than 1000, bigger than or equals to 2000 or in between.

Smallest

Return the smallest of four given integers.

String Differences

Return a list containing a number and two characters that indicate at which position two given strings are different. The characters represent the two characters, one from each string, that are different.

String Lengths Backwards

Return a list of integers containing the lengths of strings in a given list in reverse order of the given list of strings.

Sum of Squares

Return the sum of all squared values between 1 and a given integer.

Super Anagrams

Return a boolean value indicating if a string a is a super anagram of b. Every character in b must be in a as many times as it is in b. a may contain extra characters.

Syllables

Return an integer that is the number of vowels in a given string that contains only spaces, digits and lowercase letters.

Vector Average

Return a float that is the average of all floats in a given list.

Vectors Summed

Return a list of integers that is the element-wise sum of two given lists of integers of equal size.

Wallis Pi

Return a float representing the product of the first n terms of the Wallis product.

Word Stats

Return a list of integers containing the number of words of the length corresponding to the index for a given string. Additionally, return an integer representing the number of sentences as well as a float representing the average number of words per sentence.

X-Word Lines

Return a string that contains the same words as a given string, but has precisely n, which is a given integer, words per line. The given string includes spaces and newlines.

A.2 Fitness Functions

The fitness functions used for experiments with the general program synthesis benchmark suite are described in Table A.1. The fitness is minimized. Therefore the optimal value that can be reached is zero. Table A.1 describes the fitness function for a single training case. The values for all training cases are aggregated by summing them up to get the fitness values, which is required for many selection operators. Lexicase selection can operate on a non-aggregated list of values, which makes it possible to have multiple fitness functions as is the case for Replace Space with Newline, Word Stats and X-Word Lines.

"Absolute difference" is the absolute difference between two numerical values. "Boolean difference" is either zero if two boolean values are the same, otherwise one. "Levenshtein distance" is used to compare strings, which calculates the number of single-character edits (insertions, deletions or substitutions) required to transform one string into another. Fitness functions comparing lists use one of the

previously mentioned functions and sum over the calculated values. Additionally, a penalty value is used if the lists are of different length. A "length difference penalty" value is multiplied by the difference in length of the two lists. The "length difference penalty" value is set differently for each problem as it depends on the maximum and minimum values that can occur as the output of a problem.

Table A.1: Fitness functions for the problems of the general program synthesis benchmark suite used in this theses.

Problem	Fitness Function
Checksum	Absolute difference between the returned character and the expected character in the ASCII table. Note: When no character data type is available, a string is used. If the string is empty, the fitness value is set to 1000. If the string contains multiple characters, the first character is compared
Collatz Numbers	Absolute difference
Compare String Lengths	Boolean difference
Count Odds	Absolute difference
Digits	Sum of absolute difference. Length difference penalty of 20
Double Letters	Levenshtein distance
Even Squares	Sum of absolute difference. Length difference penalty of 100
For Loop Index	Sum of absolute difference. Length difference penalty of 2000
Grade	Absolute difference between the returned character and the expected character in the ASCII table. A penalty of 6 is used if no value has been set or if the returned string contains more than one character
Last Index of Zero	Absolute difference
Median	Boolean difference
Mirror Image	Boolean difference
Negative To Zero	Sum of absolute difference. Length difference penalty of 5000
Number IO	Absolute difference
Pig Latin	Levenshtein distance
Replace Space with Newline	Levenshtein distance for string comparison; Absolute difference for the number of non-whitespace characters
Scrabble Score	Absolute difference
Small Or Large	Levenshtein distance
Smallest	Boolean difference
String Differences	Has not been used, see Section 4.3.1
String Lengths Backwards	Sum of absolute difference. Length difference penalty of 50
Sum of Squares	Absolute difference
Super Anagrams	Boolean difference
Syllables	Absolute difference

Vector Average	Absolute difference
Vectors Summed	Sum of absolute difference. Length difference penalty of 10000
Wallis Pi	Absolute difference
Word Stats	Levenshtein distance between the integer lists after converting to strings; Absolute difference for the integer representing the number of sentences; Absolute difference for the float representing the average number of words per sentence
X-Word Lines	Levenshtein distance; Absolute difference of number of newlines; Sum of the absolute difference of number of words in each line

Appendix B

Program Synthesis Grammars

The following section explains how grammars are combined automatically followed by the grammars used within this thesis for all experimental chapters about program synthesis, except for Chapter 6. The grammars for Chapter 6 are in Appendix C. Finally, Section B.11 shows the code for the protected methods used within the grammars.

The grammars contain some unnecessary line breaks due to the page width.

B.1 Automatic Grammar Combination

The grammars used for the grammar design approach are split up in smaller ones, one for each available data type plus an additional grammar for defining some basic structure. Depending on the problem and its data types required, grammars can be automatically combined containing any or even all smaller grammars. Only structure.bnf must always be used.

The following grammars may define the same non-terminal symbols multiple times or even contain empty non-terminal symbols. Empty non-terminal symbols are just placeholders to make it easier to understand the grammars. For example, structure.bnf contains a `<for>` non-terminal with no productions, but shows that *for* loops can be used. The `<for>` non-terminal is defined in multiple grammars like string.bnf, list_bool.bnf, list_float.bnf etc. When combining those grammars, all the productions which are defined with the same non-terminal are put as productions in a single non-terminal symbol with the same name.

Although a grammar usually only defines functionality for a single data type, some functions require multiple data types, e.g. `<string>'['<int>':]'`, or return a different data type, e.g. <int> ::= 'len('<string>')'. Such functions are put in the grammar they fit best. When grammars are then combined, having functions that require multiple data types can lead to the issue that a certain type should not be added to the final grammar. For example, the grammar string.bnf contains multiple functions that would require the integer data type. If the final grammar should only contain bool and string as data types, integer is not available, but the final grammar contains functions that require the non-terminal `<int>`. Therefore, a clean up is done after combining the grammar, by removing productions that contain non-terminals that are not available in the grammar as well as removing non-terminals which do not contain any productions as the combined grammar should not contain any placeholders any more.

Finally, the grammars shown in this chapter contain two additional special placeholders as terminals, "forCounter%" and "loopBreak%". These terminals are used in the G3P system and are replaced before executing the final code. "forCounter%" is replaced by a variable with the same name, just instead of the "%" sign, a number is used. This can be helpful if a loop should do as many iterations as a list has elements, but not use the values of the list. The second special placeholder "loopBreak%" is used to break loops, if too many iterations have been used, e.g. in case of infinite loops. "loopBreak%" can either be replaced in the same way as "forCounter%" by replacing the "%" sign with a number to break loops individually or by replacing it with the same name without the "%" sign to use a maximum amount of iterations for all loops, as has been done in this thesis. The variable "loopBreakConst" has to be set before executing any other code to define the maximum number of iterations either for each individual loop or all loops combined.

Another technical detail is "stop.value", which is a boolean variable in shared memory to be able to stop the execution of loops from another process, if the maximal allowed execution time has been used up.

B.2 structure.bnf

```
// ****************************************************************
// Grammar containing the python structure
// ****************************************************************

<predefined> ::= <code>

<code> ::= <code><statement>'\n' | <statement>'\n'

<statement> ::= <simple_stmt> | <compound_stmt>
<simple_stmt> ::= <call> | <assign>
<compound_stmt> ::= <for>

<call> ::= // call statements
<assign> ::= // assign statements

<for> ::= // for statements depend on lists or use of range

<loop-header> ::= 'loopBreak% = 0\n'
// stop.value is set by outer code to break immediately
<loop-block> ::= ':{:\n'<code>'\nif loopBreak% > loopBreakConst or
                 stop.value:{:\nbreak\n:}loopBreak% += 1\n:}'

// for integer and float
<number> ::= <number><num> | <num>
<num> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<comp_op> ::= '<'|'>'|'=='|'>='|'<='|'!='|'is'|'is not'

// for string
<string_const_part> ::= <string_const_part><string_literal>
                      | <string_literal>

// for list
<in_list_comp_op> ::= 'in'|'not in'
<list_comp_op> ::= '=='|'!='
```

B.3 bool.bnf

```
// ****************************************************************
// bool grammar
// ****************************************************************

<bool_var> ::= // placholder for variable names

<assign> ::= <bool_assign>
<bool_assign> ::= <bool_var>' = '<bool>

<bool> ::= <bool_var> | <bool_const>
         | <bool_pre> <bool>
         | '(' <bool> <bool_op> <bool> ')'

<bool_op> ::= 'and' | 'or'
<bool_pre> ::= 'not'

<bool_const> ::= 'True' | 'False'

// structure only need if bool is used
<compound_stmt> ::= <if> | <while>

<if> ::= <if-then> | <if-then><else>
<if-then> ::= 'if '<bool><block>
<else> ::= 'else'<block>
<block> ::= ':{:\n'<code>':}'

<while> ::= <loop-header>'while '<bool><loop-block>
```

B.4 float.bnf

```
// ****************************************************************
// float grammar
// ****************************************************************

<float_var> ::= // placholder for variable names

<assign> ::= <float_assign>
<float_assign> ::= <float_var> '=' <float>
```

```
                | <float_var> <arith_assign> <float>

<float> ::= <float_var> | <float_const>
        | <arith_prefix><float>
        | '(' <float> <arith_ops> <float> ')'
        | <float_arith_ops_protected>'('<float>','<float>')'
        | 'math.ceil('<float>')' | 'math.floor('<float>')'
        | 'round('<float>')'
        | 'min('<float>', '<float>')'
        | 'max('<float>', '<float>')'
        | 'abs('<float>')'

<float_const> ::= <number>'.'<number>

<float_arith_ops_protected> ::= 'div' | 'divInt' | 'mod'

// Return int
// have to call int() for python 2.7.5
<int> ::= 'int(math.ceil('<float>'))'
        | 'int(math.floor('<float>'))' | 'int(round('<float>'))'
        | 'int('<float>')'

// Return bool
<bool> ::= <float> <comp_op> <float> | <int> <comp_op> <float>
         | <float> <comp_op> <int>
```

B.5 int.bnf

```
// ****************************************************************
// int grammar
// ****************************************************************

<int_var> ::= // placholder for variable names

<assign> ::= <int_assign>
<int_assign> ::= <int_var> '=' <int>
               | <int_var> <arith_assign> <int>

<int> ::= <int_var> | <int_const>
        | <arith_prefix><int>
        | '(' <int> <arith_ops> <int> ')'
```

```
        | <int_arith_ops_protected>'('<int>','<int>')'
        | 'min('<int>', '<int>')'
        | 'max('<int>', '<int>')'
        | 'abs('<int>')'

// to avoid problems with leading zeros in python int
<int_const> ::= 'int('<number>'.0)'

<arith_assign> ::= <arith_ops>'='
<arith_ops> ::= '+'|'-'|'*'

<int_arith_ops_protected> ::= 'divInt' | 'mod'
<arith_prefix> ::= '+'|'-'

// Return bool
<bool> ::= <int> <comp_op> <int>

// Return float
<float> ::= <int>

// Return string
<string> ::= 'saveChr('<int>')'
```

B.6 string.bnf

```
// ****************************************************************
// string grammar
// ****************************************************************

<string_var> ::= // placholder for variable names

<assign> ::= <string_assign>
<string_assign> ::= <string_var>' = '<string>

<string> ::= <string_var> | <string_const> | <string_slice>
           | <getStringIndexCall>
           | '('<string>' + '<string>')'
           | <string>'.lstrip()' | <string>'.rstrip()'
           | <string>'.strip()' | <string>'.lstrip('<string>')'
           | <string>'.rstrip('<string>')'
           | <string>'.strip('<string>')'
```

```
          | <string>'.capitalize()'

<string_slice> ::= <string>'['<int>':'<int>']'
                 | <string>'[:'<int>']' | <string>'['<int>':]'
<getStringIndexCall> ::= 'getCharFromString('<string>', '<int>')'

<string_const> ::= "'"<string_const_part>"'"
// <string_const_part> in structure(_tree).bnf
<string_literal> ::= '' | '\\n' | '\\t' | ' ' | '!' | '"' | '#'
                   | '$' | '%' | '&' | "\\'" | '(' | ')' | '*'
                   | '+' | ',' | '-' | '.' | '/' | '0' | '1' | '2'
                   | '3' | '4' | '5' | '6' | '7' | '8' | '9' | ':'
                   | ';' | '<' | '=' | '>' | '?' | '@' | 'A' | 'B'
                   | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J'
                   | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R'
                   | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
                   | '[' | '\\\\' | ']' | '^' | '_' | '`' | 'a'
                   | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
                   | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q'
                   | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y'
                   | 'z' | '{' | '|' | '}'

// Returns int
<int>  ::= 'len('<string>')' | 'saveOrd('<string>')'

// Return bool
<bool> ::= <string>' in '<string> | <string>' not in '<string>
             | <string>' == '<string> | <string>' != '<string>
             | <string>'.startswith('<string>')'
             | <string>'.endswith('<string>')'
```

B.7 list_bool.bnf

```
// ****************************************************************
// list grammar for bool list
// ****************************************************************

<list_bool_var> ::= // placholder for variable names

<assign> ::= <list_bool_assign>
<list_bool_assign> ::= <list_bool_var>' = '<list_bool>
```

```
<list_bool> ::= <list_bool_var> | <list_bool_slice>
<list_bool_slice> ::= <list_bool>'['<int>':'<int>']'
                    | <list_bool>'[:'<int>']'
                    | <list_bool>'['<int>':]'

<getListIndexCall_bool> ::=
                  'getIndexBoolList('<list_bool>', '<int>')'
<setListIndexToCall_bool> ::=
                  'setListIndexTo('<list_bool>', '<int>', '<bool>')'
<deleteListItemCall_bool> ::=
                  'deleteListItem('<list_bool>', '<int>')'

<bool> ::= <getListIndexCall_bool>


// Add to
<call> ::= <list_bool_var>'.append('<bool>')'
         | <list_bool_var>'.insert('<int>','<bool>')'
         | <deleteListItemCall_bool>
         | <setListIndexToCall_bool>

<for> ::= <loop-header>'for forCounter% in '<list_bool><loop-block>
       | <loop-header>'for '<bool_var>' in '<list_bool><loop-block>

// Return int
<int> ::= 'len('<list_bool>')'

// Return bool
<bool> ::= <bool> <in_list_comp_op> <list_bool>
         | <list_bool> <list_comp_op> <list_bool>
         | <list_bool>' == []'
```

B.8 list_float.bnf

```
// ****************************************************************
// list grammar for float list
// ****************************************************************

<list_float_var> ::= // placholder for variable names
```

```
<assign> ::= <list_float_assign>
<list_float_assign> ::= <list_float_var>' = '<list_float>

<list_float> ::= <list_float_var> | <list_float_slice>
<list_float_slice> ::= <list_float>'['<int>':'<int>']'
                     | <list_float>'[:'<int>']'
                     | <list_float>'['<int>':]'

<getListIndexCall_float> ::=
               'getIndexFloatList('<list_float>', '<int>')'
<setListIndexToCall_float> ::=
               'setListIndexTo('<list_float>', '<int>', '<float>')'
<deleteListItemCall_float> ::=
               'deleteListItem('<list_float>', '<int>')'

<float> ::= <getListIndexCall_float>

// Add to
<call> ::= <list_float_var>'.append('<float>')'
         | <list_float_var>'.insert('<int>','<float>')'
         | <deleteListItemCall_float>
         | <setListIndexToCall_float>

<for> ::=
       <loop-header>'for forCounter% in '<list_float><loop-block>
     | <loop-header>'for '<float_var>' in '<list_float><loop-block>

// Return int
<int> ::= 'len('<list_float>')'

// Return bool
<bool> ::= <float> <in_list_comp_op> <list_float>
         | <list_float> <list_comp_op> <list_float>
         | <list_float>' == []'
```

B.9 list_int.bnf

```
// ****************************************************************
// list grammar for int list
// ****************************************************************
```

```
<list_int_var> ::= // placholder for variable names

<assign> ::= <list_int_assign>
<list_int_assign> ::= <list_int_var>' = '<list_int>

<list_int> ::= <list_int_var> | <list_int_range> | <list_int_slice>
<list_int_slice> ::= <list_int>'['<int>':'<int>']'
                   | <list_int>'[:'<int>']'
                   | <list_int>'['<int>':]'
<list_int_range> ::= 'list(saveRange('<int>','<int>'))'

<getListIndexCall_int> ::= 'getIndexIntList('<list_int>', '<int>')'
<setListIndexToCall_int> ::=
                   'setListIndexTo('<list_int>', '<int>', '<int>')'
<deleteListItemCall_int> ::=
                   'deleteListItem('<list_int>', '<int>')'

// Add to
<call> ::= <list_int_var>'.append('<int>')'
         | <list_int_var>'.insert('<int>','<int>')'
         | <deleteListItemCall_int>
         | <setListIndexToCall_int>

<for> ::= <loop-header>'for forCounter% in '<list_int><loop-block>
        | <loop-header>'for '<int_var>' in '<list_int><loop-block>

// Return int
<int> ::= <getListIndexCall_int>
            | 'len('<list_int>')'

// Return bool
<bool> ::= <int> <in_list_comp_op> <list_int>
         | <list_int> <list_comp_op> <list_int>
         | <list_int>' == []'
```

B.10 list_string.bnf

```
// ****************************************************************
// list grammar for string list
// ****************************************************************
```

```
<list_string_var> ::= // placholder for variable names

<assign> ::= <list_string_assign>
<list_string_assign> ::= <list_string_var>' = '<list_string>

<list_string> ::= <list_string_var> | <list_string_slice>
                | 'saveSplit('<string>','<string>')'
<list_string_slice> ::= <list_string>'['<int>':'<int>']'
                      | <list_string>'[:'<int>']'
                      | <list_string>'['<int>':]'

<getListIndexCall_string> ::=
             'getIndexStringList('<list_string>', '<int>')'
<setListIndexToCall_string> ::=
             'setListIndexTo('<list_string>', '<int>', '<string>')'
<deleteListItemCall_string> ::=
             'deleteListItem('<list_string>', '<int>')'

// Add to
<call> ::= <list_string_var>'.append('<string>')'
         | <list_string_var>'.insert('<int>','<string>')'
         | <deleteListItemCall_string>
         | <setListIndexToCall_string>

<for> ::=
     <loop-header>'for forCounter% in '<list_string><loop-block>
   | <loop-header>'for '<string_var>' in '<list_string><loop-block>

// Return int
<int> ::= 'len('<list_string>')'

// Return string
<string> ::= <string>'.join('<list_string>')'
           | <getListIndexCall_string>

// Return bool
<bool> ::= <string> <in_list_comp_op> <list_string>
         | <list_string> <list_comp_op> <list_string>
         | <list_string>' == []'
```

B.11 Protected methods

The protected methods listed below are used in the grammars to avoid error and provide evaluation safety. Some of the methods are only used in the extended program synthesis grammars from Appendix C.

```
// ****************************************************************
# Protected methods
// ****************************************************************
import math

def div(nom, denom):
  if denom <= 0.00001:
    return nom
  else:
    return nom / denom

def divInt(nom, denom):
  if denom <= 0.00001:
    return nom
  else:
    return nom // denom

def mod(nom, denom):
  if denom <= 0.00001:
    return nom
  else:
    return nom % denom

def deleteListItem(curList, index):
  if not curList:
    return
  del curList[index % len(curList)]

def setListIndexTo(curList, index, value):
  if not curList:
    return
  curList[index % len(curList)] = value

def getIndexBoolList(curList, index):
  if not curList:
    return bool()
```

```
  return curList[index % len(curList)]

def getIndexFloatList(curList, index):
  if not curList:
    return float()
  return curList[index % len(curList)]

def getIndexIntList(curList, index):
  if not curList:
    return int()
  return curList[index % len(curList)]

def getIndexStringList(curList, index):
  if not curList:
    return str()
  return curList[index % len(curList)]

def getCharFromString(curString, index):
  if not curString:
    # with extended grammars it was changed to return ' '
    return ''
  return curString[index % len(curString)]

def saveChr(number):
    return chr(number % 128)

def saveOrd(literal):
    if len(literal) <= 0:
        return 32
    return ord(literal[0])

def saveRange(start, end):
  if end > start and abs(start - end) > 10000:
    return range(start, start + 10000)
  return range(start, end)

def setchar(s, c, i):
  if not s:
    return s
  s = list(s)
  s[i % len(s)] = c
  return ''.join(s)
```

```
chr_map = {0: chr(0), 1: chr(1)}
def int_to_chr(i):
  # 96 visible characters in ascii + space + tab + newline
  i = i % 96
  return chr_map[i] if i in chr_map else chr(i + 30)

def float_to_chr(f):
  return int_to_chr(int(f))

def saveIndex(l, i):
  if i not in l:
    return -1
  return l.index(i)

def replaceFirstElementInList(l, a, b):
  i = saveIndex(l, a)
  if i < 0:
    return l
  l[i] = b
  return l

def saveSplit(s, sep):
  if not sep:
    return s.split()
  return s.split(sep)
// ************************************************************
```

Appendix C

Extended Program Synthesis Grammars

The grammars contain some unnecessary line breaks due to the page width.

C.1 structure.bnf

```
// ****************************************************************
// Grammar containing the python structure
// ****************************************************************

<predefined> ::= <code>

<code> ::= <code><statement>'\n' | <statement>'\n'

<statement> ::= <simple_stmt> | <compound_stmt>
<simple_stmt> ::= <call> | <assign>
<compound_stmt> ::= <for>

<call> ::= // call statements
<assign> ::= // assign statements

<for> ::= // for statements depend on lists or use of range

<loop-header> ::= 'loopBreak% = 0\n'
// stop.value is set by outer code to break immediately
<loop-block> ::= ':{:\n'<code>'\nif loopBreak% > loopBreakConst or
                 stop.value:{:\nbreak\n:}loopBreak% += 1\n:}'
```

```
// for integer and float
<number> ::= <number><num> | <num>
<num> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<comp_op> ::= '<'|'>'|'=='|'>='|'<='|'!='|'is'|'is not'

// for string
<string_const_part> ::= <string_const_part><string_literal>
                      | <string_literal>

// for list
<in_list_comp_op> ::= 'in'|'not in'
<list_comp_op> ::= '=='|'!='
```

C.2 bool.bnf

```
// ****************************************************************
// bool grammar
// ****************************************************************

<bool_var> ::= // placholder for variable names

<assign> ::= <bool_assign>
<bool_assign> ::= <bool_var>' = '<bool>

<bool> ::= <bool_var> | <bool_const>
         | '('<bool_pre> <bool>')'
         | '('<bool> <bool_op> <bool>')'
         // boolean_invert_first_then_and
         | '( not' <bool> 'and' <bool>')'
         // boolean_invert_second_then_and
         | '('<bool> 'and not' <bool>')'

<bool_op> ::= 'and' | 'or' | '^' | '=='
<bool_pre> ::= 'not'

<bool_const> ::= 'True' | 'False'

// structure only need if bool is used
<compound_stmt> ::= <if> | <while>
```

```
<if> ::= <if-then> | <if-then><else>
<if-then> ::= 'if '<bool><block>
<else> ::= 'else'<block>
<block> ::= ':{:\n'<code>':}'

<while> ::= <loop-header>'while '<bool><loop-block>

// Return int
<int> ::= 'int('<bool>')'

// Return float
<float> ::= 'float('<bool>')'

// Return string
<string> ::= 'str('<bool>')'
```

C.3 float.bnf

```
// ****************************************************************
// float grammar
// ****************************************************************

<float_var> ::= // placholder for variable names

<assign> ::= <float_assign>
<float_assign> ::= <float_var> '=' <float>
                 | <float_var> <arith_assign> <float>

<float> ::= <float_var> | <float_const>
        | <arith_prefix><float>
        | '('<float> <arith_ops> <float>')'
        | <float_arith_ops_protected>'('<float>','<float>')'
        | 'math.ceil('<float>')' | 'math.floor('<float>')'
        | 'round('<float>')'
        | 'min('<float>', '<float>')'
        | 'max('<float>', '<float>')'
        | 'abs('<float>')'
        | 'math.sin('<float>')' | 'math.cos('<float>')'
        | 'math.tan('<float>')'
        | '('<float> '+ 1)' | '('<float> '- 1)'
```

```
<float_const> ::= <number>'.'<number>

<float_arith_ops_protected> ::= 'div' | 'divInt' | 'mod'

// Return int
// have to call int() for python 2.7.5
<int> ::= 'int(math.ceil('<float>'))'
        | 'int(math.floor('<float>'))' | 'int(round('<float>'))'
        | 'int('<float>')'

// Return bool
<bool> ::= '('<float> <comp_op> <float>')' | 'bool('<float>')'

// Return char
<char> ::= 'float_to_chr('<float>')'

// Return string
<string> ::= 'str('<float>')'
```

C.4 int.bnf

```
// ****************************************************************
// int grammar
// ****************************************************************

<int_var> ::= // placholder for variable names

<assign> ::= <int_assign>
<int_assign> ::= <int_var> '=' <int>
               | <int_var> <arith_assign> <int>

<int> ::= <int_var> | <int_const>
        | <arith_prefix><int>
        | '(' <int> <arith_ops> <int> ')'
        | <int_arith_ops_protected>'('<int>','<int>')'
        | 'min('<int>', '<int>')'
        | 'max('<int>', '<int>')'
        | 'abs('<int>')'
        | '('<int> '+ 1)' | '('<int> '- 1)'
```

```
// to avoid problems with leading zeros in python int
<int_const> ::= 'int('<number>'.0)'

<arith_assign> ::= <arith_ops>'='
<arith_ops> ::= '+'|'-'|'*'
// ** raises ZeroDivision error when 'a = 0; a ** ~a'
// |'**'

<int_arith_ops_protected> ::= 'divInt' | 'mod'
<arith_prefix> ::= '+'|'-'
//|'~'

// Add to
<for> ::= <loop-header>'for forCounter% in
                                saveRange(0, '<int>')'<loop-block>
        | <loop-header>'for '<int_var>' in
                                saveRange(0, '<int>')'<loop-block>
        | <loop-header>'for forCounter% in
                            saveRange('<int>', '<int>')'<loop-block>
        | <loop-header>'for '<int_var>' in
                            saveRange('<int>', '<int>')'<loop-block>

// Return bool
<bool> ::= '('<int> <comp_op> <int>')' | 'bool('<int>')'

// Return float
<float> ::= <int>

// Return char
<char> ::= 'int_to_chr('<int>')'

// Return string
<string> ::= 'str('<int>')'
```

C.5 char.bnf

```
// ****************************************************************
// char grammar
// ****************************************************************
```

```
<char_var> ::= // placholder for variable names

<assign> ::= <char_assign>
<char_assign> ::= <char_var>' = '<char>

<char> ::= <char_var> | "'"<char_literal>"'"

<char_literal> ::= '\\n' | '\\t' | ' ' | '!' | '"' | '#' | '$'
                 | '%' | '&' | "\\'" | '(' | ')' | '*' | '+' | ','
                 | '-' | '.' | '/' | '0' | '1' | '2' | '3' | '4'
                 | '5' | '6' | '7' | '8' | '9' | ':' | ';' | '<'
                 | '=' | '>' | '?' | '@' | 'A' | 'B' | 'C' | 'D'
                 | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L'
                 | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T'
                 | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | '[' | '\\\\'
                 | ']' | '^' | '_' | '`' | 'a' | 'b' | 'c' | 'd'
                 | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l'
                 | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'
                 | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | '{' | '|'
                 | '}'

// Return string
<string> ::= <char>

// Return bool
<bool> ::= <char>'.isdigit()' | <char>'.isspace()'
         | <char>'.isalpha()' | '('<char>' == '<char>')'

// Return int
<int> ::= 'ord('<char>')'
```

C.6 string.bnf

```
// ****************************************************************
// string grammar
// ****************************************************************

<string_var> ::= // placholder for variable names

<assign> ::= <string_assign>
<string_assign> ::= <string_var>' = '<string>
```

```
<string> ::= <string_var> | <string_const> | <string_slice>
           | '('<string>' + '<string>')'
           | <string>'.lstrip()' | <string>'.rstrip()'
           | <string>'.strip()' | <string>'.lstrip('<string>')'
           | <string>'.rstrip('<string>')'
           | <string>'.strip('<string>')'
           | <string>'.capitalize()'
           | "''.join(reversed("<string>"))" // reverse
           | <string>'.replace('<string>','<string>', 1)'
           | <string>'.replace('<string>','<string>')'
           | <string>'.replace('<char>','<char>', 1)'
           | <string>'.replace('<char>','<char>')'
           | <string>'.replace('<char>',"")'
           | 'setchar('<string>','<char>','<int>')'

<string_slice> ::= <string>'['<int>':'<int>']'
                 | <string>'[:'<int>']' | <string>'['<int>':]'
                 | <string>'[1:]' | <string>'[:1]'

<string_const> ::= "'"<string_const_part>"'"
// <string_const_part> in structure(_tree).bnf
<string_literal> ::= '' | '\\n' | '\\t' | ' ' | '!' | '"' | '#'
                   | '$' | '%' | '&' | "\\'" | '(' | ')' | '*'
                   | '+' | ',' | '-' | '.' | '/' | '0' | '1' | '2'
                   | '3' | '4' | '5' | '6' | '7' | '8' | '9' | ':'
                   | ';' | '<' | '=' | '>' | '?' | '@' | 'A' | 'B'
                   | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J'
                   | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R'
                   | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
                   | '[' | '\\\\' | ']' | '^' | '_' | '`' | 'a'
                   | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
                   | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q'
                   | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y'
                   | 'z' | '{' | '|' | '}'

// Add to
<for> ::= <loop-header>'for forCounter% in '<string><loop-block>
        | <loop-header>'for '<char_var>' in '<string><loop-block>
        | <loop-header>'for '<string_var>' in '<string><loop-block>
        | <loop-header>'for '<string_var>' in
                          '<string>'.strip().split()'<loop-block>
        | <loop-header>'for '<string_var>' in
```

```
            saveSplit('<string>'.strip(), '<string>')'<loop-block>

// Returns int
<int>  ::= 'len('<string>')' | <string>'.count('<char>')'
         | <string>'.count('<string>')'

// Return bool
<bool> ::= '('<string>' in '<string>')'
         | '('<string>' not in '<string>')'
         | '('<string>' == '<string>')'
         | '('<string>' != '<string>')'
         | <string>'.startswith('<string>')'
         | <string>'.endswith('<string>')'
         | '(not '<string>')'  // is empty

// Return char
<char> ::= 'getCharFromString('<string>', '<int>')'

// Return list_string
<list_string> ::= <string>'.strip().split()'
                | 'saveSplit('<string>'.strip(), '<string>')'
```

C.7 list_bool.bnf

```
// ****************************************************************
// list grammar for bool list
// ****************************************************************

<list_bool_var> ::= // placholder for variable names

<assign> ::= <list_bool_assign>
<list_bool_assign> ::= <list_bool_var>' = '<list_bool>

<list_bool> ::= <list_bool_var> | <list_bool_slice>
     | <list_bool> + <list_bool>
     | 'list(reversed('<list_bool>'))'
     | '[x if x == '<bool>' else '<bool>' for x in '<list_bool>']'
     | 'replaceFirstElementInList('<list_bool>','<bool>','<bool>')'
     | '[x for x in '<list_bool>' if x == '<bool>']'
```

```
<list_bool_slice> ::= <list_bool>'['<int>':'<int>']'
                    | <list_bool>'[:'<int>']'
                    | <list_bool>'['<int>':]'
                    | <list_bool>'[1:]' | <list_bool>'[:1]'
// Return bool
<bool> ::= 'getIndexBoolList('<list_bool>', '<int>')'
         | 'getIndexBoolList('<list_bool>', 0)'
         | 'getIndexBoolList('<list_bool>', -1)'
         | '('<bool> <in_list_comp_op> <list_bool>')'
         | '('<list_bool> <list_comp_op> <list_bool>')'
         | '('<list_bool>' == [])'

// Add to
<call> ::= <list_bool_var>'.insert('<int>','<bool>')'
         | <list_bool_var>'.insert(0,'<bool>')'
         | <list_bool_var>'.append('<bool>')'
         | 'deleteListItem('<list_bool>', '<int>')'
         | 'setListIndexTo('<list_bool>', '<int>', '<bool>')'

<for> ::= <loop-header>'for forCounter% in '<list_bool><loop-block>
       | <loop-header>'for '<bool_var>' in '<list_bool><loop-block>

// Return int
<int> ::= 'len('<list_bool>')'
        | 'saveIndex('<list_bool>', '<bool>')'
        | <list_bool>'.count('<bool>')'
```

C.8 list_float.bnf

```
// *************************************************************
// list grammar for float list
// *************************************************************

<list_float_var> ::= // placholder for variable names

<assign> ::= <list_float_assign>
<list_float_assign> ::= <list_float_var>' = '<list_float>

<list_float> ::= <list_float_var> | <list_float_slice>
  | 'list(reversed('<list_float>'))'
  | '[x if x == '<float>' else '<float>' for x in '<list_float>']'
```

```
  | 'replaceFirstElementInList('<list_float>','<float>','<float>')'
  | '[x for x in '<list_float>' if x == '<float>']'
<list_float_slice> ::= <list_float>'['<int>':'<int>']'
                     | <list_float>'[:'<int>']'
                     | <list_float>'['<int>':]'
                     | <list_float>'[1:]' | <list_float>'[:1]'

<getListIndexCall_float> ::=
               'getIndexFloatList('<list_float>', '<int>')'
<setListIndexToCall_float> ::=
               'setListIndexTo('<list_float>', '<int>', '<float>')'
<deleteListItemCall_float> ::=
               'deleteListItem('<list_float>', '<int>')'

// Return float
<float> ::= <getListIndexCall_float>
          | 'getIndexFloatList('<list_float>', 0)'
          | 'getIndexFloatList('<list_float>', -1)'

// Add to

<call> ::= <list_float_var>'.append('<float>')'
         | <list_float_var>'.insert('<int>','<float>')'
         | <list_float_var>'.insert(0,'<float>')'
         | <deleteListItemCall_float>
         | <setListIndexToCall_float>

<for> ::=
       <loop-header>'for forCounter% in '<list_float><loop-block>
     | <loop-header>'for '<float_var>' in '<list_float><loop-block>

// Return int
<int> ::= 'len('<list_float>')'
        | 'saveIndex('<list_float>', '<float>')'
        | <list_float>'.count('<float>')'

// Return bool
<bool> ::= '('<float> <in_list_comp_op> <list_float>')'
         | '('<list_float> <list_comp_op> <list_float>')'
         | '('<list_float>' == [])'
```

C.9 list_int.bnf

```
// ****************************************************************
// list grammar for int list
// ****************************************************************

<list_int_var> ::= // placholder for variable names

<assign> ::= <list_int_assign>
<list_int_assign> ::= <list_int_var>' = '<list_int>

<list_int> ::= <list_int_var> | <list_int_range>
        | <list_int_slice>
        | 'list(reversed('<list_int>'))'
        | '[x if x == '<int>' else '<int>' for x in '<list_int>']'
        | 'replaceFirstElementInList('<list_int>','<int>','<int>')'
        | '[x for x in '<list_int>' if x == '<int>']'
<list_int_slice> ::= <list_int>'['<int>':'<int>']'
                   | <list_int>'[:'<int>']'
                   | <list_int>'['<int>':]'
                   | <list_int>'[1:]' | <list_int>'[:1]'
<list_int_range> ::= 'list(saveRange('<int>','<int>'))'

<getListIndexCall_int> ::=
                   'getIndexIntList('<list_int>', '<int>')'
<setListIndexToCall_int> ::=
                   'setListIndexTo('<list_int>', '<int>', '<int>')'
<deleteListItemCall_int> ::=
                   'deleteListItem('<list_int>', '<int>')'

// Add to
<call> ::= <list_int_var>'.append('<int>')'
         | <list_int_var>'.insert('<int>','<int>')'
         | <list_int_var>'.insert(0,'<int>')'
         | <deleteListItemCall_int>
         | <setListIndexToCall_int>

<for> ::= <loop-header>'for forCounter% in '<list_int><loop-block>
        | <loop-header>'for '<int_var>' in '<list_int><loop-block>

// Return int
<int> ::= <getListIndexCall_int>
```

```
        | 'len('<list_int>')'
        | 'getIndexIntList('<list_int>', 0)'
        | 'getIndexIntList('<list_int>', -1)'
        | 'saveIndex('<list_int>', '<int>')'
        | <list_int>'.count('<int>')'

// Return bool

<bool> ::= '('<int> <in_list_comp_op> <list_int>')'
         | '('<list_int> <list_comp_op> <list_int>')'
         | '('<list_int>' == [])'
```

C.10 list_string.bnf

```
// ****************************************************************
// list grammar for string list
// ****************************************************************

<list_string_var> ::= // placholder for variable names

<assign> ::= <list_string_assign>
<list_string_assign> ::= <list_string_var>' = '<list_string>

<list_string> ::= <list_string_var> | <list_string_slice>
    | 'saveSplit('<string>','<string>')'
    | 'list(reversed('<list_string>'))'
    | '[x if x == '<string>' else '<string>'
                                        for x in '<list_string>']'
    | 'replaceFirstElementInList('<list_string>','<string>','
                                                    <string>')'
    | '[x for x in '<list_string>' if x == '<string>']'
<list_string_slice> ::= <list_string>'['<int>':'<int>']'
                      | <list_string>'[:'<int>']'
                      | <list_string>'['<int>':]'
                      | <list_string>'[1:]' | <list_string>'[:1]'

<getListIndexCall_string> ::=
             'getIndexStringList('<list_string>', '<int>')'
<setListIndexToCall_string> ::=
             'setListIndexTo('<list_string>', '<int>', '<string>')'
<deleteListItemCall_string> ::=
```

```
            'deleteListItem('<list_string>', '<int>')'

<call> ::= <list_string_var>'.insert('<int>','<string>')'
         | <deleteListItemCall_string>
         | <setListIndexToCall_string>

// Add to
<call> ::= <list_string_var>'.append('<string>')'
         | <list_string_var>'.insert('<int>','<string>')'
         | <list_string_var>'.insert(0,'<string>')'
         | <deleteListItemCall_string>
         | <setListIndexToCall_string>

<for> ::=
     <loop-header>'for forCounter% in '<list_string><loop-block>
   | <loop-header>'for '<string_var>' in '<list_string><loop-block>

// Return int
<int> ::= 'len('<list_string>')'
        | 'saveIndex('<list_string>', '<string>')'
        | <list_string>'.count('<string>')'

// Return string
<string> ::= <string>'.join('<list_string>')'
           | <getListIndexCall_string>
           | 'getIndexStringList('<list_string>', 0)'
           | 'getIndexStringList('<list_string>', -1)'

// Return bool
<bool> ::= '('<string> <in_list_comp_op> <list_string>')'
         | '('<list_string> <list_comp_op> <list_string>')'
         | '('<list_string>' == [])'
```

Appendix D

Plots for Effective Semantic Mutation for Program Synthesis

The figures on the following pages contain additional plots for Chapter 7 for Effective Semantic Mutation for Program Synthesis (ESMPS). The plots have been omitted from the chapter as they are very similar to the ones for the semantic crossover version, but put in the appendix for the sake of completeness.

Ad Figure D.1: In case of mutation the new subtree is created randomly, therefore mutation can always takes place, in contrast to crossover were it can happen that no subtree from the second parent fits in a selected position from the first parent.

D.1 Semantic Measure Used with ESMPS

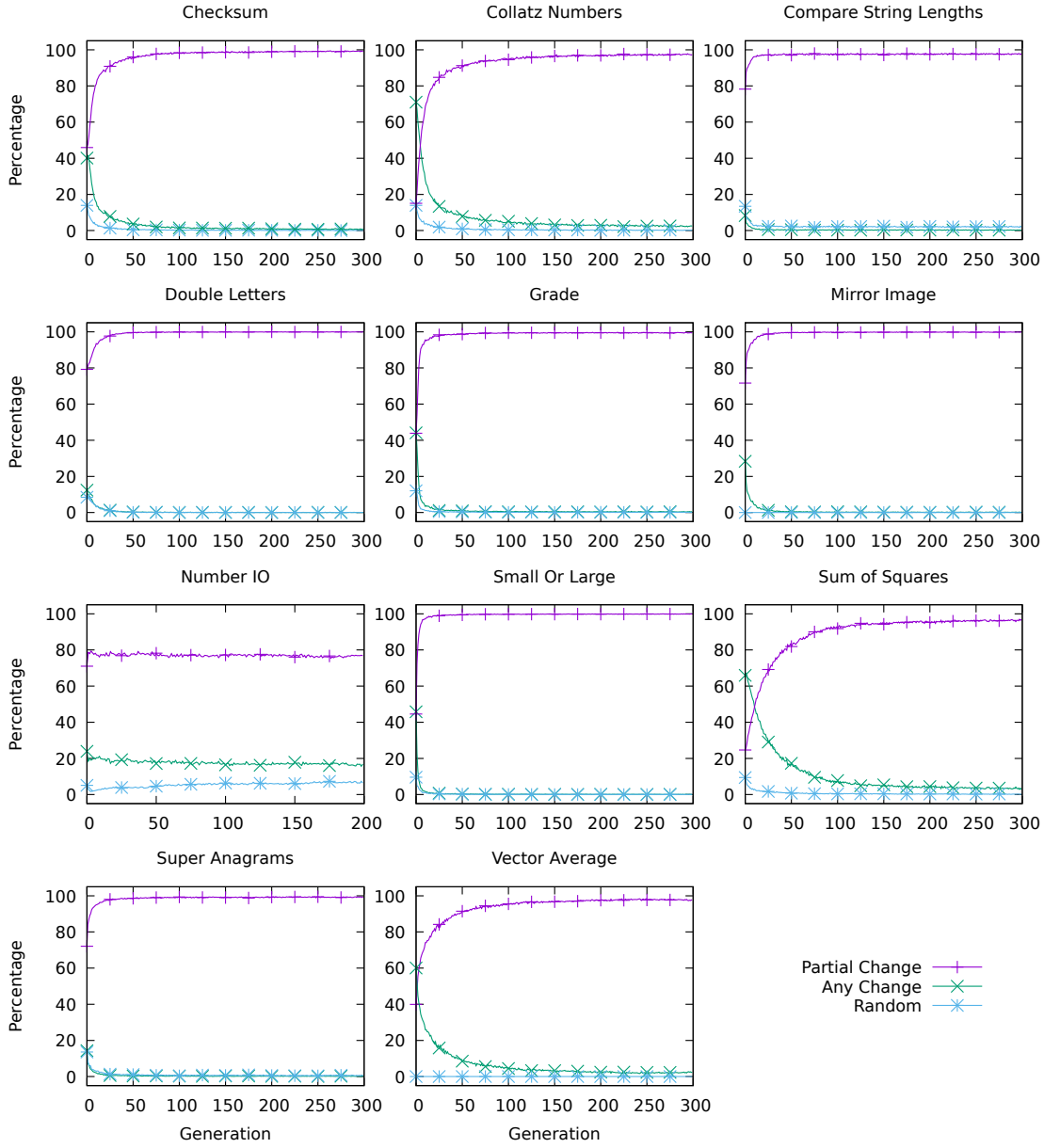


Figure D.1: Percentage of semantic measure used during mutation over generations. "Partial change" is the semantic measure that is used first. If no subtree for mutation is found, "Any change" is used before falling back to "Random" mutation.

D.2 Number of Tries for ESMPS

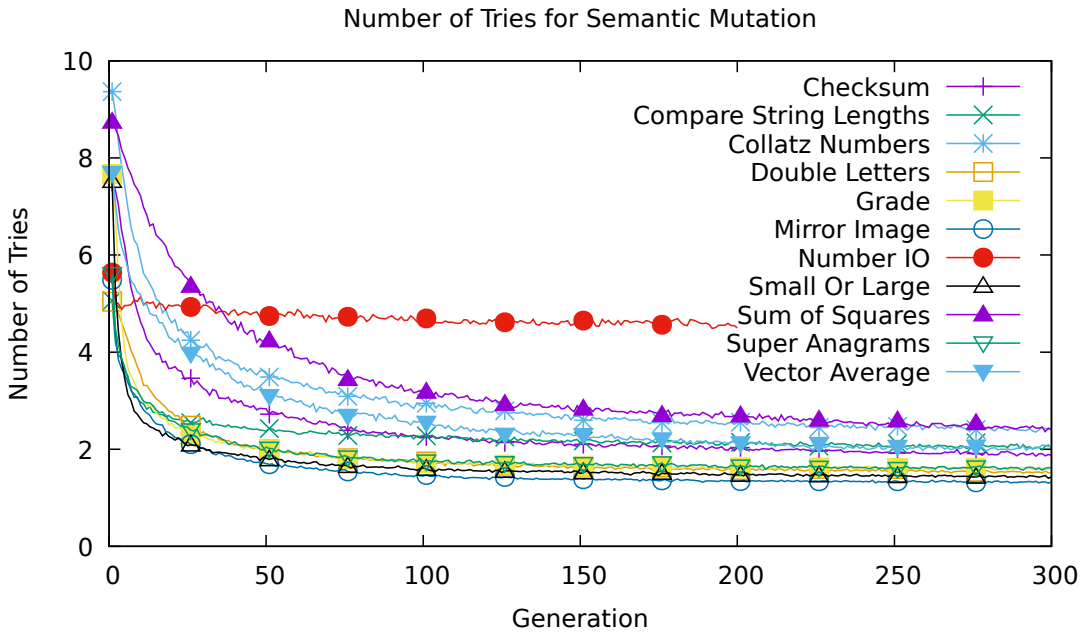


Figure D.2: Average number of tries for creating a subtree that has a "Partial Change" compared to the selected subtree with mutation.

D.3 Percentage of Semantically Different with ESMPS

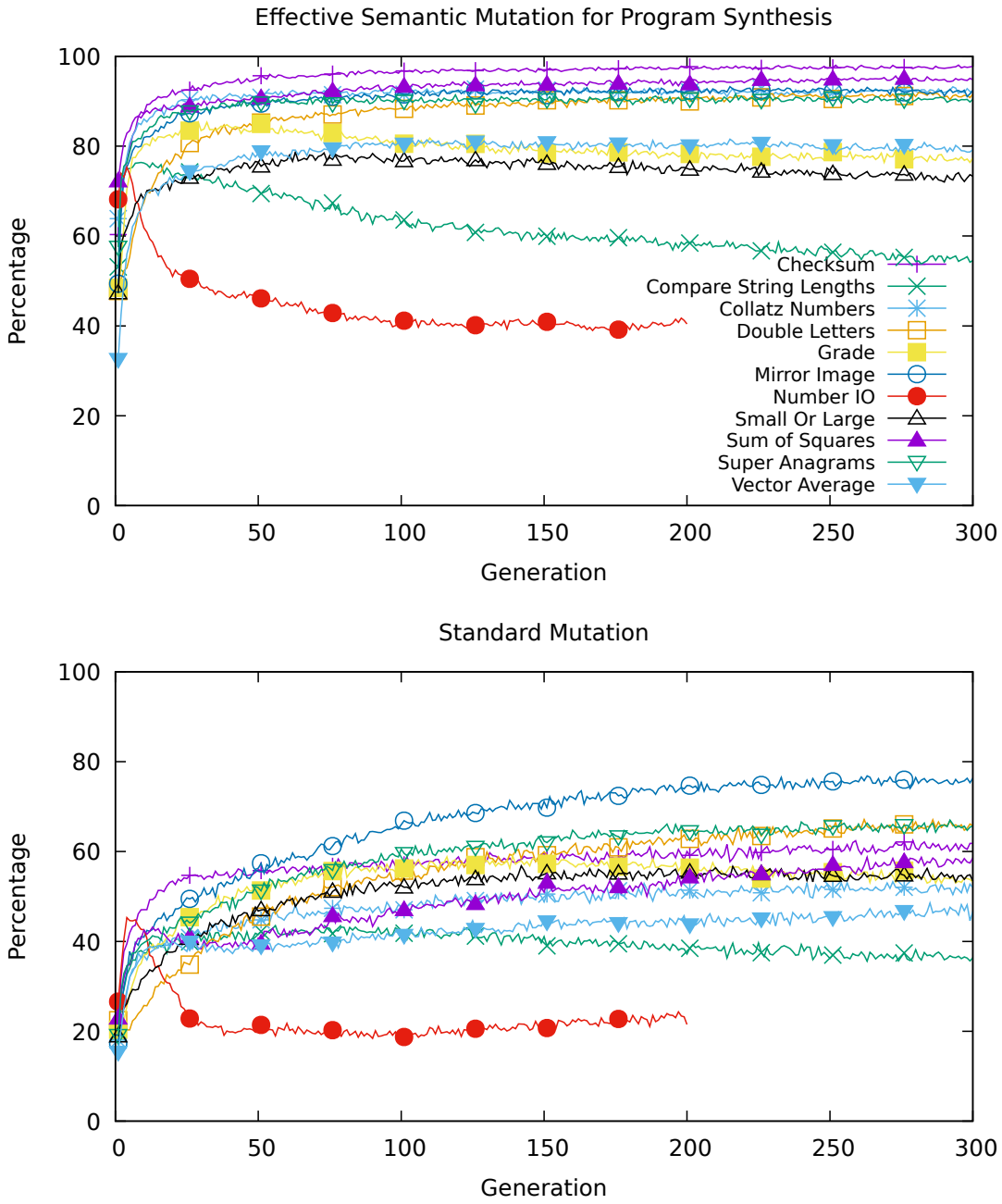


Figure D.3: Percentage of children semantically different from their rooted parent with ESMPS (top) and standard mutation (bottom).

D.4 Percentage of Fitter Children with ESMPS

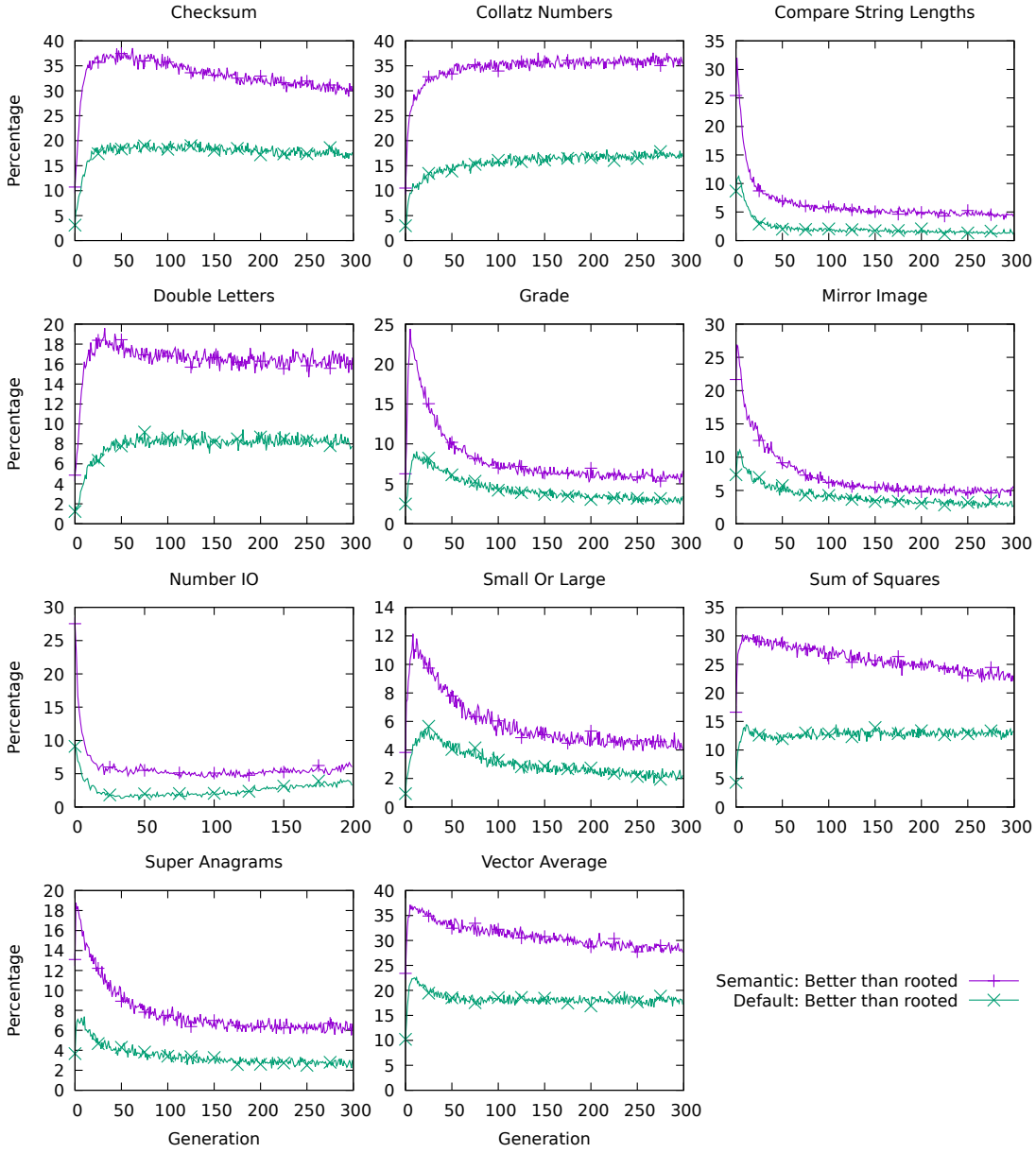


Figure D.4: Percentage of children that are better than their rooted parent over generations created with ESMPS ("Semantic") and standard mutation ("Default").

Appendix E

Grammar Design Pattern Solutions

Solutions shown here have passed all generated and hard-coded training as well as test cases. This does not prove their correctness, but not all possible inputs can be tested.

To increase readability of the code parts that are the same for every problem have been removed. That includes the initialization of temporary variables (`i0 = int();...`, `b0 = bool();...` etc.) and the variables `loopBreakConst`, `loopBreak` and `rec_counter` in the initialization, which are used to stop iterations of loops and recursion. Additionally, empty lines have been omitted as well. Otherwise, the code has not been modified in any way, although much code could be removed in many cases.

Solutions are taken from the first experiment ordered chronologically by chapter in the thesis that found a successful solution to a problem using lexicase selection. If multiple solutions to a problem have been found, one is picked at random. Of the 28 problems tackled from the general program synthesis benchmark suite, String Differences has not yet been tried, 21 problems have been solved at least once. The code of 16 solutions are from Chapter 4, code for Mirror Image is from Section 4.4.4 and Pig Latin, Replace Space with Newline, Syllables, and X-Word Lines are from Chapter 6.

The code contains some unnecessary line breaks due to the page width.

E.1 Compare String Lengths

```
def evolve(in0, in1, in2):
  res0 = bool()
  i2 = len(in0)
  if len(in1) < len(in2):
    b1 = True
    while False:
      b1 = b2
      if loopBreak > loopBreakConst or stop.value:
        break
      loopBreak += 1
    b2 = True
    res0 = in1[i2:] != s2
```

E.2 Count Odds

```
def evolve(in0):
  res0 = int()
  for i1 in in0:
    if res0 <= +mod(res0,len(li2)):
      deleteListItem(li2, i0)
      if res0 <= +mod(res0,len(li2)):
        if not +getIndexIntList(in0, i2) <= i0:
          if not getIndexIntList(li2, ( i2 + int(4.0) )) <=
          mod(i1,divInt(int(3.0),res0)):
            deleteListItem(li2, i1)
          li2.append(( i0 + i1 ))
          deleteListItem(li2, res0)
        li1.insert(i0,( i1 + res0 ))
    li1.insert(mod(abs(int(0.0)),( len(li2) + int(0.0) )),res0)
    li2.append(len(list(saveRange(i2,getIndexIntList(li2,
    int(4.0))))))
    res0 = len(li2[mod(( i1 * i1 ),max(int(0.0), int(4.0))):])
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  li2.append(i2)
  li2.append(len(list(saveRange(i2,getIndexIntList(li1,
  len(li1))))))
```

```
  deleteListItem(li2[mod(( i1 + i1 ),max(( i1 + int(4.0) ),
  int(4.0))):], res0)
  return res0
```

E.3 Even Squares

```
def evolve(in0):
  res0 = []
  while abs(( ( len(res0) * abs(( ( len(li1) *
  getIndexIntList(li1, len(res0)) ) + getIndexIntList(res0,
  len(res0)) )) ) + abs(( ( len(li1) * getIndexIntList(li1,
  len(res0)) ) + int(4.0) )) )) < in0:
    b1 = False
    res0.insert(len(res0),abs(( ( len(res0) * abs(( ( len(li1) *
    getIndexIntList(li1, len(res0)) ) + int(4.0) )) ) + abs(( (
    len(li1) * getIndexIntList(li1, len(res0)) ) + int(4.0) )) )))
    res0 = li1
    for forCounter0 in res0:
      b1 = ( int(9.0) + int(0.0) ) is not int(94.0)

      if loopBreak > loopBreakConst or stop.value:
        break
      loopBreak += 1
    b1 = ( int(9.0) + int(0.0) ) is not int(94.0)
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  return res0
```

E.4 For Loop Index

```
def evolve(in0, in1, in2):
  res0 = []
  res0.append(in0)
  for i0 in list(saveRange(in1,in0)):
    res0.insert(getIndexIntList(res0, min(i0, max(int(31.0),
    i0))),in0)
    if loopBreak > loopBreakConst or stop.value:
      break
```

```
    loopBreak += 1
  for i0 in list(saveRange(in0,in1)):
    for in0 in list(saveRange(in0,i0))[in2:int(771.0)]:
      res0.append(in0)
      if loopBreak > loopBreakConst or stop.value:
        break
      loopBreak += 1
    for i0 in list(saveRange(i0,int(771.0)))[i1:int(2.0)]:
      for in0 in list(saveRange(in0,i0))[in2:int(1171.0)]:
        res0.append(in0)
        if loopBreak > loopBreakConst or stop.value:
          break
        loopBreak += 1
      for in0 in list(saveRange(int(2.0),i1)):
        for in0 in list(saveRange(in0,int(3.0)))[in2:int(7.0)]:
          res0.append(in0)
          if loopBreak > loopBreakConst or stop.value:
            break
          loopBreak += 1
        for i0 in list(saveRange(in1,int(31.0))):
          li1.insert(int(11.0),len(li0))
          if loopBreak > loopBreakConst or stop.value:
            break
          loopBreak += 1
        if loopBreak > loopBreakConst or stop.value:
          break
        loopBreak += 1
      if loopBreak > loopBreakConst or stop.value:
        break
      loopBreak += 1
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  return res0
```

E.5 Grade

```
def evolve(in0, in1, in2, in3, in4):
  res0 = str()
  res0 = 'B'
  while False:
```

```
    res0 = s0
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  while in0 <= abs(abs(in4)):
    res0 = 'A'
    while False:
      s0 = 'C'
      if loopBreak > loopBreakConst or stop.value:
        break
      loopBreak += 1
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  b2 = in1 <= abs(abs(in4))
  while not b2:
    res0 = 'C'
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  i2 -= i0
  b2 = ( not in4 <= abs(abs(in4)) or in2 <= abs(abs(abs(in4))) )
  while not b2:
    res0 = 'D'
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  b2 = ( not in4 <= abs(abs(in4)) or in3 <= abs(abs(abs(in4))) )
  while not b2:
    res0 = 'F'
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  return res0
```

E.6 Last Index of Zero

```
def evolve(in0):
  res0 = int()
  for res0 in in0:
    li0.insert(( res0 * res0 ),len(li0))
```

```
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  res0 = getIndexIntList(li0, len(in0))
  return res0
```

E.7 Median

```
def evolve(in0, in1, in2):
  res0 = int()
  i0 = max(min(max(in0, in2), divInt(in1,+min(abs(i1),
  int(3.0)))), min(in0, divInt(min(mod(divInt(in2,i2),i0), (
  max(i2, in0) * min(( in0 - divInt(-abs(i1),max(i1, i1)) ),
  +abs(in0)) )),+min(abs(i1), int(3.0)))))
  b1 = b1
  b1 = b2
  while not False:
    res0 = i0
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  return res0
```

E.8 Mirror Image

```
def evolve(in0, in1):
  res0 = bool()
  li0 = li1
  li1.insert(-int(9.0),getIndexIntList(in0, len(li2)))
  res0 = li2 == in1
  for i2 in in0:
    res0 = in1[-len(in0[divInt(getIndexIntList(in1,
    divInt(divInt(getIndexIntList(in1,
    max(getIndexIntList(in0[--getIndexIntList(in1, i2):],
    len(in0)), int(9.0))),getIndexIntList(in1,
    getIndexIntList(in0,
    len(in1)))),+len(in1[divInt(getIndexIntList(in1,
    +i2),getIndexIntList(in0,
    len(li0))):][len(li0):][:getIndexIntList(in1,
```

```
    len(in1))])))),getIndexIntList(in0, getIndexIntList(in1,
    divInt(getIndexIntList(in1, getIndexIntList(in0,
    int(2.0))),len(in0))))):][:i1][:len(li1)]):] == li1
    li1.insert(-len(li0),getIndexIntList(in0, len(li0)))
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  return res0
```

E.9 Negative To Zero

```
def evolve(in0):
  res0 = []
  if False:
    li0 = li2
  else:
    for i2 in in0:
      res0.append(min(abs(len(list(saveRange(min(-min(divInt(int(7.
      0),divInt(max(i2, i2),abs(i2))), max(max(i0, i1), i1)),
      divInt(i2,divInt(i2,i2))),( min(max(mod(abs(i2),i0),
      int(4.0)), ( int(4.0) - ( i1 * -min(i0, i1) ) )) +
      max(-min(-abs(i2), i2), i1) ))))), max(max(i0, i2), i1)))
      if loopBreak > loopBreakConst or stop.value:
        break
      loopBreak += 1
  i2 = mod(getIndexIntList(in0, getIndexIntList(li0,
  getIndexIntList(li0, max(max(i0, i0), i1)))),abs(divInt(i2,i2)))
  if not list(saveRange((
  mod(getIndexIntList(list(saveRange(divInt(max(i2,
  int(7.0)),abs(i2)),abs(i1))),
  i1),abs(max(+getIndexIntList(list(saveRange(i1,i0)), i2), i2)))
  + len(li0) ),i2)) == li0:
    in0.append(i0)
  else:
    li2.append(i1)
    if True:
      in0.append(i0)
    else:
      in0.append(i0)
  return res0
```

E.10 Number IO

As the other solutions this one was picked at random. Many other solutions found for Number IO contain more bloat code.

```
def evolve(in0, in1):
  res0 = float()
  res0 = ( in0 + in1 )
```

E.11 Pig Latin

```
def evolve(in0):
  res0 = str()
  for in0 in saveSplit(in0.strip(), c1.lstrip()):
    for in0 in saveSplit(in0.strip(), c1.lstrip()):
      for in0 in saveSplit(in0.strip(), c1.lstrip()):
        for in0 in saveSplit(in0.strip(), '}'.lstrip()):
          for res0 in saveSplit((res0 + setchar((s0 +
          ((in0.replace('}',"").lstrip(s0.capitalize()) +
          getCharFromString((in0.replace(int_to_chr(len(in0)),
          int_to_chr(len(s1)), 1) +
          in0.capitalize().capitalize().replace('}',"").lstrip(
          ''.join(reversed('aeiou'))).lstrip()).lstrip(in0).replace
          (int_to_chr(i0),'}', 1), int('aeiou'.startswith(''.join(
          reversed('aeiou'))))).lstrip()).lstrip((in0.lstrip(in0.
          replace('}',"").lstrip(''.join(reversed((in0.lstrip(in0.
          replace(getCharFromString(''.join(reversed(s0)).lstrip(),
          len(''.join(reversed('aeiou')).replace(getCharFromString(
          ''.join(reversed(in0)).lstrip(), int(in0.startswith(in0.
          replace(c1,"")))),"").replace('}',"")),"").lstrip((in0 +
          in0).lstrip(in0.replace(getCharFromString(c1.lstrip().
          lstrip(), len((in0 + s1))),"").lstrip(''.join(reversed(
          'aeiou'))).replace(getCharFromString(in0.capitalize(),
          len(in0.replace('}',"").lstrip('aeiou'))),""))).replace(
          int_to_chr(len(in0)),"")) + s1)))).replace(
          getCharFromString(c1.lstrip(), i0),"")) + s1)).lstrip(
          'aeiou') + 'ay')),getCharFromString(s1.replace(
          int_to_chr(i0),int_to_chr(len(in0)), 1), i0),len(s1)).
          capitalize()).strip(), 'aeiou'):
            c1 = c1
            if loopBreak > loopBreakConst or stop.value:
```

```
            break
          loopBreak += 1
        if loopBreak > loopBreakConst or stop.value:
          break
        loopBreak += 1
      if loopBreak > loopBreakConst or stop.value:
        break
      loopBreak += 1
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  if loopBreak > loopBreakConst or stop.value:
    break
  loopBreak += 1
return res0
```

E.12 Replace Space with Newline

```
def evolve(in0):
  res0 = str(); res1 = int()
  for res1 in saveRange(i0, ((in0.strip().replace('\n',"").rstrip()
  + s1).replace('\n',"").replace(c0,"").rstrip() +
  s1).replace('\t',"").count(s0)):
    res0 = in0.replace(c0,'\n'.replace(in0.rstrip(),s1.strip(s0),
    1)).replace(c0,'\n'.replace(res0,'\n'.replace(in0.strip().
    replace(c0,""),in0.replace(c0,'\n', 1).replace(c0,""), 1), 1),
    1).replace('\n'.replace(s1.rstrip(),in0.replace(c0,'\n'.
    replace(in0.rstrip(),s1.strip(s0), 1), 1),
    1).strip(),'\n'.replace(s1.rstrip(),in0.replace(c0,'\n'.
    replace(in0.rstrip(),s1.strip(s0), 1), 1),
    1).strip().replace(c0,'\n'.replace(in0.rstrip(),'\n'.strip(s0),
    1), 1), 1).replace(c0,'\n'.replace(res0,s1.rstrip(), 1),
    1).replace(c0,"")
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  return res0, res1
```

E.13 Scrabble Score

Scrabble Score has a global variable `scrabblescore` that contains the correct scrabble value for every character.

```
def evolve(in0):
  global scrabblescore
  res0 = int()
  li1.append(i2)
  for i0 in list(saveRange(mod(res0,res0),max(+len(s2),
  mod(len(in0.rstrip(s2).strip(s2).lstrip()),
  len(scrabblescore))))):
    scrabblescore.append(i2)
    res0 += getIndexIntList(scrabblescore[saveOrd(s2):],
    divInt(saveOrd(getCharFromString(in0.strip(in0.rstrip(
    getCharFromString(in0, saveOrd(in0)))[:divInt(int(7.0),abs((
    max(int(3.0), divInt(abs(+len(scrabblescore)),max(i2, i2))) -
    saveOrd(s2) )))]).lstrip(), ( i2 - len(scrabblescore)
    )).lstrip().lstrip().capitalize().lstrip().rstrip()),len(
    getCharFromString(in0.strip(in0.rstrip(getCharFromString(in0,
    saveOrd(in0)))[:divInt(int(7.0),abs(( max(int(3.0),
    divInt(abs(+len(scrabblescore)),max(i2, i2))) - saveOrd(s2)
    )))]).lstrip(), ( i0 - i0 )).lstrip().strip()))
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  res0 += i2
  li2 = scrabblescore
  scrabblescore.append(i2)
  res0 += getIndexIntList(scrabblescore,
  divInt(saveOrd((in0.strip().lstrip()[len(saveChr(i2).lstrip())
  :int(3.0)].capitalize() +
  in0.lstrip().capitalize().lstrip().rstrip().capitalize().
  rstrip())),len(getCharFromString(in0, res0))))
  return res0
```

E.14 Small Or Large

```
def evolve(in0):
  res0 = str()
  s2 = s0
```

```
  if int(410.0) > ( ( in0 - int(896.0) ) - int(694.0) ):
    s2 = s2.rstrip('large').strip(res0.strip()).strip('large'.
    rstrip()).rstrip(((s0.lstrip().strip() + s0.lstrip()) +
    'large')).lstrip()
    if int(896.0) > ( in0 - int(104.0) ):
      b1 = ( not divInt(int(4.0),int(8.0)) is not int(640.0) and
      True )
      res0 = 'small'
    else:
      s2 = s2.rstrip('large').strip(res0.strip()).strip('large'.
      rstrip()).rstrip((('small'.strip() + s0.lstrip()) +
      'large')).lstrip()
  else:
    b1 = b1
    res0 = 'large'
  i1 *= -int(82.0)
  b1 = True
  return res0
```

E.15 Smallest

```
def evolve(in0, in1, in2, in3):
  res0 = int()
  if b2:
    res0 = -+int(1119.0)
  res0 += min(( abs(min(max(+int(69.0), mod(( max(mod(int(4.0),+
  in0), mod(int(49.0),abs(abs(( max(abs(min(in3, abs(in1))),
  +divInt(( in2 + ( i0 * i0 ) ),res0)) - abs(max(in3, min(in0,
  in3))) ))))) - abs(divInt(max(min(max(min(max(int(51.0), min(in0,
  in3)), abs(abs(i1))), ( -( i0 + min(in1, in1) ) + i0 )), int(9.0)
  ), min(min(abs(+i0), i1), abs(max(( --divInt(abs(i1),in2) -
  mod(i1,res0) ), max(abs(in0), i0))))),( int(6.0) * abs(abs(( in1
  - divInt(in2,in0) ))) ))) ),++i1)), res0)) + min(max(in1, in1),
  min(in2, min(min(in1, in1), min(in0, in3)))) ), max(min(i1,
  int(2.0)), divInt(abs(in0),i0)))
  return res0
```

E.16 String Lengths Backwards

```
def evolve(in0):
  res0 = []
  for forCounter0 in in0:
    ls2.append(s0)
    li2 = res0[:i0]
    res0.append(len(getIndexStringList(in0[( i2 - max(len(ls2),
    len(ls2)) ):], abs((
    len('Kch&zFz"={C{t5&z=&z"=?{&="h&zt{&z={=z"="5)cv58F?t58ch&"={h
    h&F5\tch&?5\'FhYh&={ztz?"=C') * len(ls2) )))))
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  return res0
```

E.17 Sum of Squares

```
def evolve(in0):
  res0 = int()
  res0 = divInt(( +max(in0, abs(int(6.0))) + abs(abs(max(i2, (
  abs(max(i1, ( divInt(+( in0 + ( +min(abs(in0), int(3.0)) +
  divInt(in0,i2) ) ),i2) * in0 ))) * in0 )))) ),int(6.0))
  i0 -= i0
  if i2 > min(( ( +( i2 - ( mod(+( +in0 + in0 ),+( int(979.0) +
  int(95880.0) )) * int(640.0) ) ) + -i2 ) + ( +int(0.0) - i2 ) ),
  abs(int(97.0))):
    i0 -= abs(int(6.0))
  else:
    res0 = divInt(( abs(int(970497.0)) + ( int(6.0) + int(511889.0)
    ) ),abs(abs(int(6.0))))
  return res0
```

E.18 Syllables

```
def evolve(in0):
  res0 = int()
  for in0 in saveSplit((in0.replace(''.join(reversed(s2)),'+'.
```

```
rstrip()) + s2).strip(), s2.rstrip()):
  for in0 in saveSplit((in0.replace(''.join(reversed(s2)),'+'.
  rstrip()) + s2).strip(), s2.rstrip()):
    for in0 in saveSplit(((in0 +
    s1.replace(in0.capitalize(),str(int(4.0)).replace(setchar(s2,
    c2,int(4.0)).replace(''.join(reversed(''.join(reversed(s2)).
    rstrip())),s1),(in0 + c2).rstrip())))) + s0).strip(),
    getCharFromString(''.join(reversed(getCharFromString(
    getCharFromString(''.join(reversed(getCharFromString(''.join
    (reversed(in0.rstrip('aeiou').capitalize().capitalize().
    rstrip().capitalize())), max(int(4.0), ('aeiou'.capitalize().
    count(getCharFromString(in0, int(4.0))) + 1))))), res0),
    max(int(4.0), int(3.0))).capitalize())),
    len('o'.rstrip())).rstrip()):
      for in0 in saveSplit(in0.rstrip().strip(), 'aeiou'):
        for in0 in saveSplit((in0 + ''.join(reversed(''.join(
        reversed('aeiou'.rstrip(in0).capitalize()))).rstrip())
        )).strip(), getCharFromString(in0.rstrip('aeiou'),
        min(int(4.0), (res0 + 1))).rstrip()):
          res0 += +s2.count(s1.replace(in0.capitalize().replace(
          in0,s0.replace(c1,int_to_chr(i1)), 1).rstrip().
          replace('{',""),'*'))
          if loopBreak > loopBreakConst or stop.value:
            break
          loopBreak += 1
        if loopBreak > loopBreakConst or stop.value:
          break
        loopBreak += 1
      if loopBreak > loopBreakConst or stop.value:
        break
      loopBreak += 1
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  if loopBreak > loopBreakConst or stop.value:
    break
  loopBreak += 1
return res0
```

E.19 Vector Average

```
def evolve(in0):
  res0 = float()
  lf1.append(2.204)
  for res0 in in0:
    lf1.append(+min(-2.27, i1))
    for res0 in lf0[:int(2.0)]:
      for res0 in lf0[:int(3.0)]:
        for forCounter0 in in0:
          i1 += int(113.0)
          res0 -= div(+min(-getIndexFloatList(in0, max(int(339.0),
          i1)), i1),math.ceil(div(len(in0),min(int(round(f2)),
          9.723))))
          lf1.append(div(+min(min(int(round(f2)), 2.834),
          int(2389.0)),min(int(round(math.ceil(18.2))),
          math.ceil(div(int(round(math.ceil(4.3))),f2)))))
          if loopBreak > loopBreakConst or stop.value:
            break
          loopBreak += 1
        if loopBreak > loopBreakConst or stop.value:
          break
        loopBreak += 1
      if loopBreak > loopBreakConst or stop.value:
        break
      loopBreak += 1
    for f2 in lf1:
      for f2 in lf1:
        lf0.append(f0)
        if loopBreak > loopBreakConst or stop.value:
          break
        loopBreak += 1
      if loopBreak > loopBreakConst or stop.value:
        break
      loopBreak += 1
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  return res0
```

E.20 Vectors Summed

```
def evolve(in0, in1):
  res0 = []
  for i2 in in1:
    li0 = li0
    i1 = min(( i0 - i2 ), ( i0 - i2 ))
    res0.append(( mod(getIndexIntList(in0, ( i1 - ( ( i1 - int(0.0)
    ) - len(res0) ) )),int(0.0)) - i1 ))
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  return res0
```

E.21 X-Word Lines

```
def evolve(in0, in1):
  res0 = str()
  for s0 in in0.strip().split():
    s2 = ('\n' + s1.replace(res0,res0, 1)).replace(
    getCharFromString(setchar(c2.strip(setchar(res0,'\n',len(c2)).
    replace(c0,c0)),'2',res0.lstrip().rstrip().count(c2)).
    replace(';',';'), (in1 - 1)),"")
    res0 = s2.replace(getCharFromString(setchar(s2,'\n',int(1.0)).
    replace(setchar(s2,getCharFromString(s2.replace(c2.strip(res0).
    replace('B',""),s0, 1), +( (res0.lstrip().rstrip().count(c2) +
    1) - (res0.count(c2) + 1) )),(+int((not False)) + 1)).replace(
    '\n',""),c2, 1), (max(max(min((setchar(setchar(res0,c2,len(
    s2.capitalize()+1))).replace('\n',c2)[:in1],'\n',+(setchar(res0,
    '2','\n'.lstrip().rstrip(s2).count(c2)).replace('\n',c2).count(
    c2) + 1)).capitalize().lstrip().count('\n') + 1), (+(
    (res0.lstrip().count(c1) + 1) - in1 ) + 1)),
    '\n'.lstrip().rstrip(s2).count(c2)), s2.rstrip().count(c0)) -
    1)),c2, 1).replace(s2,s2, 1).replace(setchar(s2,'\n',(+int(
    False) + 1)).replace('\n',""),(res0 + s0), 1).lstrip()
    if loopBreak > loopBreakConst or stop.value:
      break
    loopBreak += 1
  return res0
```

Bibliography

[1] T. Helmuth and L. Spector, "General program synthesis benchmark suite," in *GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, (Madrid, Spain), pp. 1039–1046, ACM, 11-15 July 2015.

[2] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, (New York, NY, USA), pp. 215–224, ACM, 2010.

[3] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, (Washington, DC, USA), pp. 364–374, IEEE Computer Society, 2009.

[4] S. O. Haraldsson, J. R. Woodward, A. E. I. Brownlee, and D. Cairns, "Exploring fitness and edit distance of mutated python programs," in *Genetic Programming* (J. McDermott, M. Castelli, L. Sekanina, E. Haasdijk, and P. García-Sánchez, eds.), (Cham), pp. 19–34, Springer International Publishing, 2017.

[5] S. O. Haraldsson, J. R. Woodward, A. E. I. Brownlee, and K. Siggeirsdottir, "Fixing bugs in your sleep: How genetic improvement became an overnight success," in *GI-2017* (J. Petke, D. R. White, W. B. Langdon, and W. Weimer, eds.), (Berlin), pp. 1513–1520, ACM, 15-19 July 2017. Best paper.

[6] T. Lau, P. Domingos, and D. S. Weld, "Learning programs from traces using version space algebra," in *Proceedings of the 2Nd International Conference*

on Knowledge Capture, K-CAP '03, (New York, NY, USA), pp. 36–43, ACM, 2003.

[7] C. Shearer, "The CRISP-DM model: The new blueprint for data mining," *Journal of Data Warehousing*, 2000.

[8] R. S. Olson, R. J. Urbanowicz, P. C. Andrews, N. A. Lavender, L. C. Kidd, and J. H. Moore, *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 – April 1, 2016, Proceedings, Part I*, ch. Automating Biomedical Data Science Through Tree-Based Pipeline Optimization, pp. 123–137. Springer International Publishing, 2016.

[9] S. Katayama, "Recent improvements of MagicHaskeller," in *Approaches and Applications of Inductive Programming*, pp. 174–193, Springer Verlag, 2010.

[10] S. Muggleton, L. De Raedt, D. Poole, I. Bratko, P. Flach, K. Inoue, and A. Srinivasan, "Ilp turns 20," *Machine Learning*, vol. 86, pp. 3–23, Jan 2012.

[11] T. A. Lau, P. Domingos, and D. S. Weld, "Version space algebra and its application to programming by demonstration," in *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, (San Francisco, CA, USA), pp. 527–534, Morgan Kaufmann Publishers Inc., 2000.

[12] D. J. Montana, "Strongly typed genetic programming," *Evol. Comput.*, vol. 3, pp. 199–230, June 1995.

[13] M. O'Neill, M. Nicolau, and A. Agapitos, "Experiments in program synthesis with grammatical evolution: A focus on integer sorting," in *Evolutionary Computation (CEC), 2014 IEEE Congress on*, pp. 1504–1511, July 2014.

[14] W. B. Langdon and M. Harman, "Optimizing existing software with genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 19, pp. 118–135, Feb 2015.

[15] J. R. Woodward and R. Bai, "Why evolution is not a good paradigm for program induction: A critique of genetic programming," in *Proceedings of the First ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, GEC '09, (New York, NY, USA), pp. 593–600, ACM, 2009.

[16] L. Vanneschi, M. Castelli, and S. Silva, "A survey of semantic methods in genetic programming," *Genetic Programming and Evolvable Machines*, vol. 15, no. 2, pp. 195–214, 2014.

[17] P. A. Whigham, *Grammatical Bias for Evolutionary Learning*. PhD thesis, University of New South Wales, New South Wales, Australia, Australia, 1996. AAI0597571.

[18] M. O'Neill and C. Ryan, *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[19] M. O'Neill and C. Ryan, "Automatic generation of caching algorithms," in *Evolutionary Algorithms in Engineering and Computer Science* (K. Miettinen, M. M. Mäkelä, P. Neittaanmäki, and J. Periaux, eds.), (Jyväskylä, Finland), pp. 127–134, John Wiley & Sons, 30 May - 3 June 1999.

[20] R. Loughran, J. McDermott, and M. O'Neill, "Tonality driven piano compositions with grammatical evolution," in *IEEE Congress on Evolutionary Computation, CEC 2015, Sendai, Japan, May 25-28, 2015*, pp. 2168–2175, IEEE, 2015.

[21] M. Fenton, C. McNally, J. Byrne, E. Hemberg, J. McDermott, and M. O'Neill, "Automatic innovative truss design using grammatical evolution," *Automation in Construction*, vol. 39, no. 0, pp. 59 – 69, 2014.

[22] J. Byrne, M. Fenton, E. Hemberg, J. McDermott, and M. O'Neill, "Optimising complex pylon structures with grammatical evolution," *Information Sciences*, no. 0, pp. –, 2014.

[23] J. Byrne, P. Cardiff, A. Brabazon, and M. O'Neill, "Evolving parametric aircraft models for design exploration and optimisation," *Neurocomputing*, vol. 142, no. 0, pp. 39 – 47, 2014. {SI} Computational Intelligence Techniques for New Product Development.

[24] E. Hemberg, L. Ho, M. O'Neill, and H. Claussen, "A comparison of grammatical genetic programming grammars for controlling femtocell network coverage," *Genetic Programming and Evolvable Machines*, vol. 14, no. 1, pp. 65–93, 2013.

[25] M. Fenton, D. Lynch, S. Kucera, H. Claussen, and M. O'Neill, "Multilayer optimization of heterogeneous networks using grammatical genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, (Berlin, Germany), pp. 3–4, ACM, 15-19 July 2017.

[26] E. Hemberg, *An Exploration of Grammars in Grammatical Evolution*. PhD thesis, University College Dublin, Ireland, 2010.

[27] E. Murphy, *An Exploration of Tree-Adjoining Grammars for Grammatical Evolution*. PhD thesis, University College Dublin, Ireland, 6 Dec. 2014.

[28] M. Nicolau, "Automatic grammar complexity reduction in grammatical evolution," in *GECCO 2004 Workshop Proceedings*, (Seattle, Washington, USA), 26-30 June 2004.

[29] M. Nicolau, M. O'Neill, and A. Brabazon, "Termination in grammatical evolution: grammar design, wrapping, and tails," in *2012 IEEE Congress on Evolutionary Computation*, pp. 1–8, June 2012.

[30] L. Beadle and C. Johnson, "Semantically driven crossover in genetic programming," in *Proceedings of the IEEE World Congress on Computational Intelligence* (J. Wang, ed.), (Hong Kong), pp. 111–116, IEEE Computational Intelligence Society, IEEE Press, 1-6 June 2008.

[31] N. F. McPhee, B. Ohs, and T. Hutchison, *Semantic Building Blocks in Genetic Programming*, pp. 134–145. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[32] Q. U. Nguyen, X. H. Nguyen, and M. O'Neill, "Semantic aware crossover for genetic programming: The case for real-valued function regression," in *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009* (L. Vanneschi, S. Gustafson, A. Moraglio, I. De Falco, and M. Ebner, eds.), vol. 5481 of *LNCS*, (Tuebingen), pp. 292–302, Springer, Apr. 15-17 2009.

[33] Q. U. Nguyen, X. H. Nguyen, M. O'Neill, R. I. McKay, and E. Galván-López, "Semantically-based crossover in genetic programming: application

to real-valued symbolic regression," *Genetic Programming and Evolvable Machines*, vol. 12, pp. 91–119, June 2011.

[34] Q. U. Nguyen, X. H. Nguyen, M. O'Neill, R. I. McKay, and D. N. Phong, "On the roles of semantic locality of crossover in genetic programming," *Information Sciences*, vol. 235, pp. 195–213, 20 June 2013.

[35] L. Beadle and C. Johnson, "Semantically driven mutation in genetic programming," in *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pp. 1336–1342, May 2009.

[36] Q. U. Nguyen, X. H. Nguyen, and M. O'Neill, "Semantics based mutation in genetic programming: The case for real-valued symbolic regression," in *15th International Conference on Soft Computing, Mendel'09* (R. Matousek and L. Nolle, eds.), (Brno, Czech Republic), pp. 73–91, June 24-26 2009.

[37] S. Forstenlechner, "GitHub repository: HeuristicLab.CFGGP: Provides context free grammar problems for HeuristicLab," 2018. [Online; accessed 07-May-2018].

[38] S. Wagner, G. Kronberger, A. Beham, M. Kommenda, A. Scheibenpflug, E. Pitzer, S. Vonolfen, M. Kofler, S. Winkler, V. Dorfer, and M. Affenzeller, *Advanced Methods and Applications in Computational Intelligence*, vol. 6 of *Topics in Intelligent Engineering and Informatics*, ch. Architecture and Design of the HeuristicLab Optimization Environment, pp. 197–261. Springer, 2014.

[39] M. Fenton, J. McDermott, D. Fagan, S. Forstenlechner, E. Hemberg, and M. O'Neill, "Ponyge2: Grammatical evolution in python," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, (Berlin, Germany), pp. 1194–1201, ACM, 2017.

[40] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, and U.-M. O'Reilly, "Genetic programming needs better benchmarks," in *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, (New York, NY, USA), pp. 791–798, ACM, 2012.

[41] D. R. White, J. Mcdermott, M. Castelli, L. Manzoni, B. W. Goldman, G. Kronberger, W. Jaśkowski, U.-M. O'Reilly, and S. Luke, "Better gp benchmarks: Community survey results and proposals," *Genetic Programming and Evolvable Machines*, vol. 14, pp. 3–29, Mar. 2013.

[42] J. Woodward, S. Martin, and J. Swan, "Benchmarks that matter for genetic programming," in *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, (New York, NY, USA), pp. 1397–1404, ACM, 2014.

[43] C. Darwin, "On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life," *John Murray, London*, pp. 1–556, Oct. 1859.

[44] L. Fogel, A. Owens, and M. Walsh, *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons, 1966.

[45] I. Rechenberg, *Evolutionsstrategie: optimierung technischer systeme nach prinzipien der biologischen evolution*. Frommann-Holzboog, 1973.

[46] H.-P. Schwefel, *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*, vol. 26 of *ISR*. Basel/Stuttgart: Birkhaeuser, 1977.

[47] N. Hansen and A. Ostermeier, "Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation," in *Proceedings of IEEE International Conference on Evolutionary Computation*, pp. 312–317, May 1996.

[48] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Cambridge, MA, USA: MIT Press, 1975.

[49] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[50] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA, USA: MIT Press, 1994.

[51] J. R. Koza, D. Andre, F. H. Bennett, and M. A. Keane, *Genetic Programming III: Darwinian Invention & Problem Solving*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 1999.

[52] J. R. Koza, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[53] K. A. De Jong, *Evolutionary Computation: A Unified Approach*. MIT press, 2006.

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

[55] J. Daida and A. Hilss, "Identifying structural mechanisms in standard genetic programming," in *Genetic and Evolutionary Computation — GECCO 2003* (E. Cantú-Paz, J. Foster, K. Deb, L. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. Potter, A. Schultz, K. Dowsland, N. Jonoska, and J. Miller, eds.), vol. 2724 of *Lecture Notes in Computer Science*, pp. 1639–1651, Springer Berlin Heidelberg, 2003.

[56] S. Luke, "Two fast tree-creation algorithms for genetic programming," *Evolutionary Computation, IEEE Transactions on*, vol. 4, pp. 274–283, Sep 2000.

[57] H. Xie and M. Zhang, "Impacts of sampling strategies in tournament selection for genetic programming," *Soft Computing*, vol. 16, pp. 615–633, Apr 2012.

[58] A. Brindle, *Genetic Algorithms for Function Optimization*. PhD thesis, University of Alberta, 1981.

[59] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1989.

[60] L. Spector, "Assessment of problem modality by differential performance of lexicase selection in genetic programming: A preliminary report," in *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '12, (New York, NY, USA), pp. 401–408, ACM, 2012.

[61] T. Helmuth, L. Spector, and J. Matheson, "Solving uncompromising problems with lexicase selection," *IEEE Transactions on Evolutionary Computation*, vol. 19, pp. 630–643, Oct 2015.

[62] W. La Cava, L. Spector, and K. Danai, "Epsilon-lexicase selection for regression," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, (New York, NY, USA), pp. 741–748, ACM, 2016.

[63] J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane, "Four problems for which a computer program evolved by genetic programming is competitive with human performance," in *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, vol. 1, pp. 1–10, IEEE Press, 1996.

[64] S. Luke and L. Spector, "A comparison of crossover and mutation in genetic programming," in *Genetic Programming 1997: Proceedings of the Second Annual Conference* (J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, eds.), (Stanford University, CA, USA), pp. 240–248, Morgan Kaufmann, 13-16 July 1997.

[65] M. O'Neill, L. Vanneschi, S. Gustafson, and W. Banzhaf, "Open issues in genetic programming," *Genetic Programming and Evolvable Machines*, vol. 11, pp. 339–363, Sep 2010.

[66] I. Dempsey, M. O'Neill, and A. Brabazon, *Foundations in Grammatical Evolution for Dynamic Environments*. Springer Publishing Company, Incorporated, 1st ed., 2009.

[67] A. Church, "Applications of recursive arithmetic to the problem of circuit synthesis," *Summaries of the Summer Institute of Symbolic Logic*, vol. 1, pp. 3–50, 1957.

[68] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959.

[69] J. R. Koza, "Darwinian invention and problem solving by means of genetic programming," in *IEEE SMC'99 Conference Proceedings. 1999*

IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.99CH37028), vol. 3, pp. 604–609 vol.3, Oct 1999.

[70] S. Gulwani, "Dimensions in program synthesis," in *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, (New York, NY, USA), pp. 13–24, ACM, 2010.

[71] J. R. Koza, "Hierarchical genetic algorithms operating on populations of computer programs," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'89, (San Francisco, CA, USA), pp. 768–774, Morgan Kaufmann Publishers Inc., 1989.

[72] D. L. Parnas, "Software aspects of strategic defense systems," *Commun. ACM*, vol. 28, pp. 1326–1335, Dec. 1985.

[73] B. A. Hockey and M. Rayner, "Using paraphrases of deep semantic representions to support regression testing in spoken dialogue systems," in *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, SETQA-NLP '09, (Stroudsburg, PA, USA), pp. 14–21, Association for Computational Linguistics, 2009.

[74] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, (New York, NY, USA), pp. 313–326, ACM, 2010.

[75] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, eds., *Watch What I Do: Programming by Demonstration*. Cambridge, MA, USA: MIT Press, 1993.

[76] N. Immerman, "Upper and lower bounds for first order expressibility," in *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*, pp. 74–82, Oct 1980.

[77] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv, "A simple inductive synthesis methodology and its applications," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages*

and Applications, OOPSLA '10, (New York, NY, USA), pp. 36–46, ACM, 2010.

[78] R. P. Nix, "Editing by example," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 600–621, Oct. 1985.

[79] C. de la Higuera, "A bibliographical study of grammatical inference," *Pattern Recogn.*, vol. 38, pp. 1332–1348, Sept. 2005.

[80] S. Katayama, "Systematic search for lambda expressions," in *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005.*, pp. 111–126, 2005.

[81] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, (New York, NY, USA), pp. 317–330, ACM, 2011.

[82] A. Solar Lezama, *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[83] S. Katayama, "Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening," in *PRICAI 2008: Trends in Artificial Intelligence* (T.-B. Ho and Z.-H. Zhou, eds.), (Berlin, Heidelberg), pp. 199–210, Springer Berlin Heidelberg, 2008.

[84] T. M. Mitchell, "Generalization as search," *Artificial Intelligence*, vol. 18, no. 2, pp. 203–226, 1982.

[85] C. Ryan, J. Collins, and M. Neill, "Grammatical evolution: Evolving programs for an arbitrary language," in *Genetic Programming* (W. Banzhaf, R. Poli, M. Schoenauer, and T. Fogarty, eds.), vol. 1391 of *Lecture Notes in Computer Science*, pp. 83–96, Springer Berlin Heidelberg, 1998.

[86] R. McKay, N. Hoai, P. Whigham, Y. Shan, and M. O'Neill, "Grammar-based genetic programming: a survey," *Genetic Programming and Evolvable Machines*, vol. 11, no. 3-4, pp. 365–396, 2010.

[87] N. Paterson, *Genetic programming with context-sensitive grammars*. PhD thesis, Saint Andrew's University, Sept. 2002.

[88] R. Cleary and M. O'Neill, "An attribute grammar decoder for the 01 multiconstrained knapsack problem," in *Evolutionary Computation in Combinatorial Optimization* (G. Raidl and J. Gottlieb, eds.), vol. 3448 of *Lecture Notes in Computer Science*, pp. 34–45, Springer Berlin Heidelberg, 2005.

[89] M. L. Wong and K. S. Leung, "An induction system that learns programs in different programming languages using genetic programming and logic grammars," in *Proceedings of 7th IEEE International Conference on Tools with Artificial Intelligence*, pp. 380–387, Nov 1995.

[90] M. L. Wong and K. S. Leung, "Evolutionary program induction directed by logic grammars," *Evol. Comput.*, vol. 5, pp. 143–180, June 1997.

[91] M. O'Neill, A. Brabazon, M. Nicolau, S. M. Garraghy, and P. Keenan, "πgrammatical evolution," in *Genetic and Evolutionary Computation – GECCO 2004* (K. Deb, ed.), (Berlin, Heidelberg), pp. 617–629, Springer Berlin Heidelberg, 2004.

[92] N. Lourenço, F. B. Pereira, and E. Costa, "Sge: A structured representation for grammatical evolution," in *Artificial Evolution* (S. Bonnevay, P. Legrand, N. Monmarché, E. Lutton, and M. Schoenauer, eds.), (Cham), pp. 136–148, Springer International Publishing, 2016.

[93] E. Medvet, "Hierarchical grammatical evolution," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, (New York, NY, USA), pp. 249–250, ACM, 2017.

[94] A. Brabazon, M. O'Neill, and S. McGarraghy, *Natural Computing Algorithms*. Springer Publishing Company, Incorporated, 1st ed., 2015.

[95] L. Spector and A. Robinson, "Genetic programming and autoconstructive evolution with the push programming language," *Genetic Programming and Evolvable Machines*, vol. 3, no. 1, pp. 7–40, 2002.

[96] T. Castle and C. G. Johnson, "Evolving high-level imperative program trees with strongly formed genetic programming," in *Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012* (A. Moraglio,

S. Silva, K. Krawiec, P. Machado, and C. Cotta, eds.), vol. 7244 of *LNCS*, (Malaga, Spain), pp. 1–12, Springer Verlag, 11-13 Apr. 2012.

[97] M. Harman, J. Krinke, J. Ren, and S. Yoo, "Search based data sensitivity analysis applied to requirement engineering," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, (New York, NY, USA), pp. 1681–1688, ACM, 2009.

[98] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, pp. 11:1–11:61, Dec. 2012.

[99] M. Harman, P. McMinn, J. T. de Souza, and S. Yoo, *Search Based Software Engineering: Techniques, Taxonomy, Tutorial*, pp. 1–59. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[100] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark, "The gismoe challenge: constructing the pareto program surface using genetic programming to find better programs (keynote paper)," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pp. 1–14, Sept 2012.

[101] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, pp. 54–72, Jan.-Feb. 2012.

[102] J. Swan, M. G. Epitropakis, and J. R. Woodward, "Gen-O-Fix: an embeddable framework for dynamic adaptive genetic improvement programming," Tech. Rep. CSM-195, Computing Science and Mathematics, University of Stirling, UK, 17 Jan. 2014.

[103] L. S. T. M. Helmuth, "Detailed problem descriptions for general program synthesis benchmark suite," tech. rep., School of Computer Science, University of Massachusetts Amherst, 2015.

[104] R. Moll, "ijava - an online interactive textbook for elementary java instruction: Demonstration," *J. Comput. Sci. Coll.*, vol. 26, pp. 55–57, June 2011.

[105] C. L. Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, pp. 1236–1256, Dec 2015.

[106] Q. U. Nguyen, *Examining Semantic Diversity and Semantic Locality of Operators in Genetic Programming.* PhD thesis, University College Dublin, Ireland, 18 July 2011.

[107] N. Dershowitz and J.-P. Jouannaud, "Rewrite systems," in *Handbook of Theoretical Computer Science Volume B: Formal Methods and Semantics* (J. van Leeuwen, ed.), ch. 6, pp. 243–320, Cambridge, MA, USA: MIT Press, 1990.

[108] E. Galván-López, B. Cody-Kenny, L. Trujillo, and A. Kattan, "Using semantics in the selection mechanism in genetic programming: A simple method for promoting semantic diversity," in *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pp. 2972–2979, June 2013.

[109] S. Forstenlechner, M. Nicolau, D. Fagan, and M. O'Neill, "Introducing semantic-clustering selection in grammatical evolution," in *GECCO 2015 Semantic Methods in Genetic Programming (SMGP'15) Workshop* (C. Johnson, K. Krawiec, A. Moraglio, and M. O'Neill, eds.), (Madrid, Spain), pp. 1277–1284, ACM, 11-15 July 2015.

[110] T. H. Chu, Q. U. Nguyen, and M. O'Neill, "Semantic tournament selection for genetic programming based on statistical analysis of error vectors," *Information Sciences*, vol. 436-437, pp. 352 – 366, 2018.

[111] K. Krawiec, "Medial crossovers for genetic programming," in *Genetic Programming* (A. Moraglio, S. Silva, K. Krawiec, P. Machado, and C. Cotta, eds.), vol. 7244 of *Lecture Notes in Computer Science*, pp. 61–72, Springer Berlin Heidelberg, 2012.

[112] K. Krawiec and T. Pawlak, "Locally geometric semantic crossover: a study on the roles of semantics and homology in recombination operators," *Genetic Programming and Evolvable Machines*, vol. 14, pp. 31–63, Mar 2013.

[113] A. Moraglio, K. Krawiec, and C. Johnson, "Geometric semantic genetic programming," in *Parallel Problem Solving from Nature - PPSN XII* (C. Coello, V. Cutello, K. Deb, S. Forrest, G. Nicosia, and M. Pavone, eds.), vol. 7491 of *Lecture Notes in Computer Science*, pp. 21–31, Springer Berlin Heidelberg, 2012.

[114] M. Castelli, S. Silva, and L. Vanneschi, "A c++ framework for geometric semantic genetic programming," *Genetic Programming and Evolvable Machines*, vol. 16, pp. 73–81, Mar 2015.

[115] J. F. B. S. Martins, L. O. V. B. Oliveira, L. F. Miranda, F. Casadei, and G. L. Pappa, "Solving the exponential growth of symbolic regression trees in geometric semantic genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, (New York, NY, USA), pp. 1151–1158, ACM, 2018.

[116] P. Orzechowski, W. La Cava, and J. H. Moore, "Where are we now?: A large benchmark study of recent symbolic regression methods," in *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, (New York, NY, USA), pp. 1183–1190, ACM, 2018.

[117] T. P. Pawlak, "Geometric semantic genetic programming is overkill," in *Genetic Programming* (M. I. Heywood, J. McDermott, M. Castelli, E. Costa, and K. Sim, eds.), (Cham), pp. 246–260, Springer International Publishing, 2016.

[118] S. Forstenlechner, M. Nicolau, D. Fagan, and M. O'Neill, "Grammar design for derivation tree based genetic programming systems," in *EuroGP 2016: Proceedings of the 19th European Conference on Genetic Programming* (M. I. Heywood, J. McDermott, M. Castelli, and E. Costa, eds.), vol. 9594 of *LNCS*, (Porto, Portugal), pp. 192–207, Springer Verlag, 30 Mar.–1 Apr. 2016.

[119] E. Murphy, E. Hemberg, M. Nicolau, M. O'Neill, and A. Brabazon, "Grammar bias and initialisation in grammar based genetic programming," in *Genetic Programming* (A. Moraglio, S. Silva, K. Krawiec, P. Machado, and C. Cotta, eds.), vol. 7244 of *Lecture Notes in Computer Science*, pp. 85–96, Springer Berlin Heidelberg, 2012.

[120] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, pp. 879–899, May 2008.

[121] J. R. Koza, I. Bennett, F.H., J. Hutchings, S. Bade, M. A. Keane, and D. Andre, "Evolving sorting networks using genetic programming and the rapidly reconfigurable xilinx 6216 -programmable gate array," in *Signals, Systems amp; Computers, 1997. Conference Record of the Thirty-First Asilomar Conference on*, vol. 1, pp. 404–410 vol.1, Nov 1997.

[122] L. Sekanina and M. Bidlo, "Evolutionary design of arbitrarily large sorting networks using development," *Genetic Programming and Evolvable Machines*, vol. 6, no. 3, pp. 319–347, 2005.

[123] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[124] M. Codish, L. Cruz-Filipe, M. Frank, and P. Schneider-Kamp, "Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten)," in *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, pp. 186–193, Nov 2014.

[125] M. Codish, L. Cruz-Filipe, T. Ehlers, M. Müller, and P. Schneider-Kamp, "Sorting networks: To the end and back again," *Journal of Computer and System Sciences*, 2016.

[126] J. McDermott, J. M. Swafford, M. Hemberg, J. Byrne, E. Hemberg, M. Fenton, C. McNally, E. Shotton, and M. O'Neill, "String-rewriting grammars for evolutionary architectural design," *Environment and Planning B: Planning and Design*, vol. 39, no. 4, pp. 713–731, 2012.

[127] I. Dempsey, M. O'Neill, and A. Brabazon, "Constant creation in grammatical evolution," *International Journal of Innovative Computing and Applications*, vol. 1, no. 1, pp. 23–38, 2007.

[128] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, "A grammar design pattern for arbitrary program synthesis problems in genetic programming," in *EuroGP 2017: Proceedings of the 20th European Conference on Genetic*

Programming (M. Castelli, J. McDermott, and L. Sekanina, eds.), vol. 10196 of *LNCS*, (Amsterdam, Netherlands), pp. 262–277, Springer Verlag, 19-21 Apr. 2017.

[129] M. O'Neill, E. Hemberg, C. Gilligan, E. Bartley, J. McDermott, and A. Brabazon, "Geva: Grammatical evolution in java," *SIGEVOlution*, vol. 3, pp. 17–22, July 2008.

[130] T. Saber, D. Fagan, D. Lynch, S. Kucera, H. Claussen, and M. O'Neill, "Multi-level grammar genetic programming for scheduling in heterogeneous networks," in *Genetic Programming* (M. Castelli, L. Sekanina, M. Zhang, S. Cagnoni, and P. García-Sánchez, eds.), (Cham), pp. 118–134, Springer International Publishing, 2018.

[131] P. J. Landin, "The next 700 programming languages," *Commun. ACM*, vol. 9, pp. 157–166, Mar. 1966.

[132] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London mathematical society*, vol. 2, no. 1, pp. 230–265, 1937.

[133] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem. a correction," *Proceedings of the London Mathematical Society*, vol. 2, no. 1, pp. 544–546, 1938.

[134] R. I. B. McKay, "Fitness sharing in genetic programming," in *Proceedings of the 2Nd Annual Conference on Genetic and Evolutionary Computation*, GECCO'00, (San Francisco, CA, USA), pp. 435–442, Morgan Kaufmann Publishers Inc., 2000.

[135] T. Helmuth, N. F. McPhee, E. Pantridge, and L. Spector, "Improving generalization of evolved programs through automatic simplification," in *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, (Berlin, Germany), pp. 937–944, ACM, 15-19 July 2017.

[136] E. Pantridge, T. Helmuth, N. F. McPhee, and L. Spector, "On the difficulty of benchmarking inductive program synthesis methods," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, (New York, NY, USA), pp. 1589–1596, ACM, 2017.

[137] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, "Towards understanding and refining the general program synthesis benchmark suite with genetic programming," in *2018 IEEE Congress on Evolutionary Computation (CEC)*, (Rio de Janeiro, Brasil), July 2018.

[138] A. L. Gaunt, M. Brockschmidt, R. Singh, N. Kushman, P. Kohli, J. Taylor, and D. Tarlow, "TerpreT: A probabilistic programming language for program induction," *CoRR*, vol. abs/1608.04428, 2016.

[139] I. Gonçalves and S. Silva, "Balancing learning and overfitting in genetic programming with interleaved sampling of training data," in *Genetic Programming* (K. Krawiec, A. Moraglio, T. Hu, A. Ş. Etaner-Uyar, and B. Hu, eds.), (Berlin, Heidelberg), pp. 73–84, Springer Berlin Heidelberg, 2013.

[140] Z. Cataltepe, Y. S. Abu-Mostafa, and M. Magdon-Ismail, "No free lunch for early stopping," *Neural Comput.*, vol. 11, pp. 995–1009, May 1999.

[141] C. Tuite, A. Agapitos, M. O'Neill, and A. Brabazon, "Early stopping criteria to counteract overfitting in genetic programming," in *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '11, (New York, NY, USA), pp. 203–204, ACM, 2011.

[142] Y. Liu and T. Khoshgoftaar, "Reducing overfitting in genetic programming models for software quality classification," in *Proceedings of the Eighth IEEE International Conference on High Assurance Systems Engineering*, HASE'04, (Washington, DC, USA), pp. 56–65, IEEE Computer Society, 2004.

[143] J. Fitzgerald, R. M. A. Azad, and C. Ryan, "A bootstrapping approach to reduce over-fitting in genetic programming," in *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '13 Companion, (New York, NY, USA), pp. 1113–1120, ACM, 2013.

[144] J. Žegklitz and P. Pošík, "Model selection and overfitting in genetic programming: Empirical study," in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, (New York, NY, USA), pp. 1527–1528, ACM, 2015.

[145] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, "Extending program synthesis grammars for grammar-guided genetic programming," in *Parallel Problem Solving from Nature – PPSN XV* (A. Auger, C. M. Fonseca, N. Lourenço, P. Machado, L. Paquete, and D. Whitley, eds.), (Coimbra, Portugal), pp. 197–208, Springer International Publishing, 2018.

[146] A. Agapitos and S. M. Lucas, "Learning recursive functions with object oriented genetic programming," in *Proceedings of the 9th European Conference on Genetic Programming* (P. Collet, M. Tomassini, M. Ebner, S. Gustafson, and A. Ekárt, eds.), vol. 3905 of *Lecture Notes in Computer Science*, (Budapest, Hungary), pp. 166–177, Springer, 10 - 12 Apr. 2006.

[147] T. Yu, "A higher-order function approach to evolve recursive programs," in *Genetic Programming Theory and Practice III*, vol. 9 of *Genetic Programming*, ch. 7, pp. 93–108, Ann Arbor: Springer, 12-14 May 2005.

[148] M. Keijzer, C. Ryan, G. Murphy, and M. Cattolico, "Undirected training of run transferable libraries," in *Proceedings of the 8th European Conference on Genetic Programming*, vol. 3447 of *Lecture Notes in Computer Science*, (Lausanne, Switzerland), pp. 361–370, Springer, 30 Mar. - 1 Apr. 2005.

[149] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, "Semantics-based crossover for program synthesis in genetic programming," in *Artificial Evolution* (E. Lutton, P. Legrand, P. Parrend, N. Monmarché, and M. Schoenauer, eds.), (Paris, France), pp. 58–71, Springer International Publishing, 2018.

[150] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, "Towards effective semantic operators for program synthesis in genetic programming," in *GECCO '18: Genetic and Evolutionary Computation Conference*, (Kyoto, Japan), ACM, 15-19 July 2018.

[151] T. M. Helmuth, *General Program Synthesis from Examples Using Genetic Programming with Parent Selection Based on Random Lexicographic Orderings of Test Cases*. PhD thesis, College of Information and Computer Sciences, University of Massachusetts Amherst, USA, Sept. 2015.

[152] T. Helmuth, N. F. McPhee, and L. Spector, "Program synthesis using uniform mutation by addition and deletion," in *Proceedings of the Genetic and*

Evolutionary Computation Conference, GECCO '18, (New York, NY, USA), pp. 1127–1134, ACM, 2018.

[153] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari, "The irace package: Iterated racing for automatic algorithm configuration," *Operations Research Perspectives*, vol. 3, pp. 43–58, 2016.

[154] T. Bartz-Beielstein, C. Lasarczyk, and M. Zaefferer, "Sequential parameter optimization," in *Proceedings Congress on Evolutionary Computation 2005 (CEC'05)*, (Edinburgh, Scotland), p. 1553, 2005.