An Exploration of Grammars in Grammatical Evolution

Erik Anders Pieter Hemberg,

M.Sc. Chalmers University of Technology

The thesis is submitted to University College Dublin

for the degree of Ph.D.

at the School of Computer Science Department and Informatics

Research Supervisors:

Dr. Michael O'Neill

Prof. Anthony Brabazon

External Examiner:

Dr. William Langdon

September 17, 2010

Abstract

The grammar in the grammar-based Genetic Programming (GP) approach of Grammatical Evolution (GE) is explored. The GE algorithm solves problems by using a grammar representation and an automated and parallel trial-and-error approach, Evolutionary Computation (EC). The search for solutions in EC is driven by evaluating each solution, selecting the fittest and replacing these into a population of solutions which are modified to further guide the search. Representations have a strong impact on the efficiency of search and by using a generative grammar domain knowledge is encoded into the population of solutions. The grammar in GE biases the search for solutions, and in combination with a linear representation this is what distinguishes GE from other GP-systems.

After a review of grammars in EC and a description of GE, several different constructions of grammars and operators for manipulating the grammars and the evolutionary algorithm are studied. The thesis goes on to study a meta-grammar GE, which allows a larger grammar with different bias. By adopting a divide-and-conquer strategy the goal is to investigate how a modular GE approach solves problems of increasing size and in dynamically changing environments. The results show some benefit from using meta-grammars in GE, for the meta-grammar Genetic Algorithm (mGGA) and they re-emphasize the grammar's impact on GE's performance.

In addition, GE and meta-grammars are more formally described. The bias, both declarative and search, arising from the use of a Context-Free Grammar representation and the constraints of GE and the mGGA are analyzed and their implications are examined. This is done by studying the effects of the mapping and operations on the input, single and multiple changes in input, as well as the preservation of output after a change. Furthermore, a matrix view of a grammar and different suggestions for measurements of grammars are investigated, in order to allow the practitioner to get an alternative view of the mapping process and of how operations work.

To my parents

Acknowledgment

I want to thank all the Natural Computation Research and Application group members, especially mentioning the directors Michael O'Neill and Anthony Brabazon, as well as Conor Gilligan, James McDermott, Jonathan Byrne, Edgar Galvan-Lopez and John-Mark Swafford. I also thank Cornelia Haeringen for proofreading.

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

Contents

List of Figures **ix**

List of Tables **xiii**

List of Grammars **xv**

Publications Arising **xviii**

1 Introduction **1**

1.1 Evolutionary Computation 2

1.1.1 Grammars - Representation 4

1.2 Research Aims 6

1.2.1 Questions 6

1.2.2 Objectives of Thesis 7

1.3 Contributions 8

1.4 Limitations 10

1.5 Thesis Overview 11

I Preliminaries - Preparing for an Exploration of Grammars in Grammatical Evolution **13**

2 Grammar Representation **15**

2.1 Grammar Definitions 15

2.1.1 Context-Free Grammar 16
2.1.2 Probabilistic Context-Free Grammar 19
2.2 Grammars in Evolutionary Computation 22
2.2.1 Grammar-based Algorithms 23
2.2.2 Measuring Grammars 29
2.2.3 Different Grammar Measures 29
2.2.4 Grammar Properties for Search 30
2.3 Summary 32

3 Description of Grammatical Evolution 33
3.1 GE Algorithm 34
3.1.1 Biological Inspiration 35
3.1.2 GE Control Flow 35
3.1.3 Grammar Mapping in GE 38
3.1.4 Background to Mapping in GE 41
3.2 Operators 47
3.2.1 Variation Operations 48
3.2.2 Selection and Replacement 49
3.3 GE in Practice 50
3.3.1 Applications 50
3.3.2 Implementing GE 51
3.4 Summary 52

II Experiments - Exploring Grammars in Grammatical Evolution 53

4 Grammar Mapping 55
4.1 Pre-, In-, Postfix Grammars 55
4.1.1 Symbolic Regression 58

4.1.2 Grammar . . . 58
4.2 Experiment . . . 59
4.2.1 Setup . . . 60
4.3 Results . . . 61
4.4 Discussion . . . 62
4.5 Summary . . . 63

5 Meta-Grammar for Automatically Defining Functions - Modularity 67
5.1 Modularity . . . 68
5.1.1 Modularity Overview . . . 68
5.1.2 EC Examples of Modularity . . . 70
5.1.3 Modularity in GP . . . 74
5.1.4 Automatically Defined Functions in GE . . . 76
5.2 Meta-Grammars and Grammatical Evolution . . . 76
5.2.1 Grammatical Evolution by Grammatical Evolution . . . 77
5.3 Experiments & Results . . . 80
5.3.1 Meta-Grammar ADF . . . 80
5.3.2 Setup . . . 80
5.3.3 Ant Trails . . . 81
5.3.4 Results - Ant Trails . . . 83
5.3.5 Symbolic Regression . . . 84
5.3.6 Results - Symbolic Regression . . . 88
5.3.7 Discussion . . . 89
5.4 Summary . . . 91

6 Meta-Grammar for Genetic Algorithms - Scalability 94
6.1 Meta-Grammar Genetic Algorithm . . . 94
6.1.1 Grammars for Bit Strings . . . 95
6.1.2 Examples of mGGA Grammars . . . 98

6.2 Scalability of the mGGA . . . 103
6.2.1 Regularity . . . 104
6.2.2 Modular Genetic Algorithm . . . 104
6.2.3 Checkerboard Problem . . . 105
6.2.4 mGGA on the Checkerboard . . . 106
6.2.5 mGGA Performance under Noisy Conditions . . . 109
6.3 Summary . . . 109

7 Altering Search Rates of the Meta- and Solution Grammars in the mGGA 112
7.1 Different Mutation Rates . . . 113
7.1.1 Setup . . . 114
7.1.2 Results . . . 115
7.2 Sampling Each Solution Grammar n Times . . . 115
7.3 Discussion . . . 118
7.3.1 Mutation Rate . . . 118
7.3.2 Length Inspection . . . 119
7.4 Summary . . . 121

8 Grammatical Bias and Use of Building Block Structures in the mGGA 127
8.1 Grammar Design . . . 128
8.2 Experiments & Results . . . 130
8.2.1 Checkerboard Results . . . 132
8.2.2 Dynamic Noisy Checkerboard Results . . . 134
8.3 Building Block Structure Usage . . . 134
8.4 Summary . . . 135

III Theory - Formalizing the Exploration of Grammars in Grammatical Evolution 143

9 Formal Description of GE and the mGGA 145

9.1 The GE Components . . . 147
9.1.1 Representation Spaces in GE . . . 156
9.1.2 Full Description of GE . . . 157
9.2 The mGGA . . . 158
9.3 Discussion . . . 159
9.3.1 The mGGA . . . 160
9.4 Summary . . . 160

10 Theory of Disruption in GE **162**
10.1 Single Change in the Chromosome . . . 163
10.1.1 Codon Change . . . 163
10.1.2 Integer Production Choice Change . . . 164
10.1.3 Change Grammar Design . . . 168
10.2 Multiple Changes in the Chromosome . . . 171
10.2.1 Crossover . . . 171
10.3 Input Change and Output Preservation . . . 172
10.3.1 Change Effects . . . 172
10.3.2 Branch Change . . . 175
10.3.3 Ripple . . . 176
10.4 Disruption in a GE Population . . . 177
10.5 The mGGA . . . 181
10.5.1 Changes in the mGGA . . . 181
10.6 Discussion . . . 181
10.6.1 *pi*-GE . . . 182
10.6.2 GE Schema . . . 183
10.7 Summary . . . 184

11 Grammar Measurements **187**
11.1 Grammar Identity and Properties . . . 188

11.1.1 Grammar Measurements for GE 189
11.1.2 GE Grammar Design and Complexity Measure 190
11.2 Probabilistic Context-Free Grammars in a Matrix Representation 191
11.2.1 Non-terminal Expectation Matrix 192
11.2.2 Terminal Expectation Matrix 196
11.3 Examples of the mGGA in Matrix Representation 197
11.3.1 Rewriting GE Grammars 197
11.3.2 Meta-Grammars in Matrix Representation 199
11.4 Discussion 201
11.5 Summary 203

IV Conclusions - Resolution of the Exploration of Grammars in Grammatical Evolution 204

12 Conclusions & Future Work 205
12.1 Thesis Summary 205
12.1.1 Contributions 208
12.1.2 Limitations of Thesis 210
12.2 Opportunities for Future Research 211

List of Definitions 215

List of Examples 216

A Example individuals 217

Bibliography 222

List of Figures

1.1 Flow of a canonical Evolutionary Algorithm 3

1.2 The parts in GE which are explored are the operators in the search algorithm and the grammar used to map the input to the output. 5

3.1 GE components 34

3.2 GE mapping flow: input and grammar are mapped to output that is evaluated and assigned a fitness 38

3.3 Example of a derivation tree that generates a word, `gjkh`, using Grammar 3.2 41

3.4 Python implementation of GE mapping. Unexpanded non-terminal symbols are put on a stack and the input is used to determine which production will be chosen from each unexpanded non-terminal. The mapping is terminated when the stack is empty or the input is used up, if there are unexpanded non-terminals when the input is used the output is set to None (invalid). Python almost reads like pseudo-code. 42

3.5 GE implemented with a GA used as a search engine. 48

3.6 Single point crossover in GE 49

3.7 Integer flip mutation in GE 49

4.1 Derivation trees mapped from the different post-, in- and prefix grammars 57

4.2 Expression trees and a plot of the function over the range 59

4.3 Best fitness results averaged over 1000 runs for pre-, in- and postfix experiments. 65

4.4 Box plots of Best Fitness and % invalids at the final generation for pre-, in- and postfix experiments. For each target to the left is infix, center is postfix and right is prefix 66

5.1 Fig. 5.1(a) shows a schematic view of the abstract concept of modularity. Fig. 5.1(b) shows a schematic view of classification of modularity. 69
5.2 An overview of the meta-grammar approach to GE. The meta-grammar generates a solution grammar, which is used to generate a candidate solution. 77
5.3 Derivation of meta-grammar and solution grammar from Grammar 5.1 . . 79
5.4 Santa Fe Ant trail, average best fitness over the runs with error bars for each generation 85
5.5 Los Altos Ant trail, average best fitness with error bars over generations . 86
5.6 Average best fitness plot with error bars over the generations for the San Mateo ant trail. A t-test confirms that the fitness differs significantly between standard GE and the other grammars for all problems in the final generation. 86
5.7 Symbolic regression polynoms 87
5.8 Plot over generations for Eq. (5.1) 91
5.9 Plot over generations for Eq. (5.2) 92
5.10 Plot over generations for dynamic functions with a period of 10. 93

6.1 An example of the mapping of a meta-grammar. Diamonds are non- terminal symbols and rectangles are terminal symbols. The numbers by the arrows are used to denote which input chooses the production from the rule. . . . 101
6.2 The original Checkerboard-pattern matching problem instances in Fig. 6.2(a) (from the top Cb_{32}, Cb_{128_0} and Cb_{512}). Fig. 6.2(b) Cb_{128_1} shows a new Checkerboard-pattern matching problem instance with a more fine-grained regularity. 106
6.3 A graph for the mGGA for Cb_{128_1} Checkerboard (1X1) is shown in 6.3(a), and the standard Cb_{128_0} instance with $p_n = 0.05$ in 6.3(b). 111

7.1 Implicit sampling of a meta-grammar GE 113
7.2 Explicit sampling of a meta-grammar GE 113
7.3 On the x-axis are the problem instances, indicated by the total number of bits, and on the y-axis the number of fitness evaluations (log-scale). 116
7.4 mGGA implicit sampling development of fitness 122
7.5 On the x-axis are the problem instances, indicated by the total number of bits, and on the y-axis the number of fitness evaluations (log-scale). 123
7.6 Note log scale on x-axis and normalized fitness on y-axis. Fig. 7.6(a) shows Cb_{32} and Fig. 7.6(b) shows Cb_{72}. The development of best fitness during the fitness evaluations for explicit sampling. 124
7.7 Average number of codons used at the end of each instance for n-samples in Fig. 7.1 and different mutation rates in Fig. 7.2. 125
7.8 Histogram over the number of combinations of the possible paths in the solution grammars and codons used for Cb_{128_0} in Fig. 7.8(a) and Cb_{200} in Fig. 7.8(b). Samples with "Infinite" value are given the max value for each setting. 126

8.1 On the x-axis is the number of fitness evaluations. On the y-axis is the normalized fitness for the different grammar versions. Fig. 8.1(a) shows Cb_{32} and Fig. 8.1(b) shows Cb_{128_0} 137
8.2 On the x-axis is the number of generations. On the y-axis is the normalized fitness for the different grammar versions for Cb_{512} 138
8.3 The appearance of solutions for a sample of 100 runs of Cb_{32}. 139
8.4 mGGA grammar bias fitness sum of errors period 10 140
8.5 In Fig. 8.5(a) the x-axis is the fitness evaluation and the y-axis is the normalized fitness and in Fig. 8.5(b) the x-axis is the period and the y-axis the sum of errors for each period for the different grammar versions 141
8.6 Use of building block structures in solution for mGGA Cb_{128_0} for Grammar 1 in Fig. 8.6(a) and using Grammar 1 and random search in Fig. 8.6(b). . . . 142

9.1 GE spaces 146
9.2 The mapping in the mGGA 161

10.1 Grammars with different numbers of rules 170
10.2 Example individual and different changes 185
10.3 Derivation tree, string schema 186

11.1 Non-terminals, rule indexes, probability indexes, unique probabilities and production choice probability 192
11.2 Grammar 2, A 200
11.3 Average length of codon use at the end point of each instance 201

List of Tables

2.1 The Chomsky Hierarchy 16

2.2 Evolutionary Algorithms that explicitly use a grammar 25

3.1 Biology and GE analogy 36

4.1 Parameters for the GE algorithm 60

4.2 p-values for the grammars on the different targets, the average best fitness and standard deviation are shown next to the grammar. *Italics* indicate a significant p-value. 61

5.1 Definitions of modules 71

5.2 Parameter settings for the GE algorithm 81

5.3 p-values for the grammars compared to the standard grammar on the different trails 84

5.4 p-values for the grammars compared to the standard grammar on the different functions 90

6.1 Parameters for the GE algorithm 107

6.2 Performance changes for the mGGA on the standard non-noisy problem instances. 108

6.3 Performance values for the mGGA on the non-noisy problem instances. . . 108

6.4 Statistics for performance of the mGGA on the Noisy Checkerboard instances for Cb_{128_0}. Success rate is the proportion of successfull solutions over the runs. 109

7.1 mGGA parameters for sampling 115

8.1 mGGA grammar bias settings 132
8.2 mGGA grammar bias performance 133
8.3 mGGA grammar bias statistics 134

9.1 Codon change and production choice type change. These are the impacts from a redundant deterministic mapping. 149
9.2 Codon change, current rule and production choice type change. 152
9.3 Codon change, production choice, current rule and next rule type change. . 153

10.1 Derivation tree change effects for the derivation $\delta = \alpha A \beta, \alpha, \beta \in V^*$ with derivation tree $D(A)$. The change effects are dependent on if the subtree changes size, then it is either a ripple or a branch 174

List of Grammars

2.1 Example of a CFG in BNF for generating a bit string 17

2.2 An example of a BNF for a bitstring of size 4 with three rules. 18

3.1 Example of a grammar for boolean expressions. `<expr>` has three production choices, `<biop>` has four production choices, `<uop>` has one production choice and `<bool>` has two production choices. 39

3.2 Example of a grammar for words. 43

4.1 Prefix grammars for Symbolic Regression, *italics* mark the difference between infix and postfix grammars. 56

4.2 Infix grammars for Symbolic Regression, *italics* mark the difference between prefix and postfix grammars. 56

4.3 Postfix grammar for Symbolic Regression, *italics* mark the difference between infix and prefix grammars. 56

5.1 Simple meta-grammar example for evolving multiple functions. Note that `<code>` and `<line>` are quoted. 78

5.2 Example Ant trail meta-grammar, adf_{mg}, is a meta-grammar that can evolve ant-trail solution grammars. 80

5.3 *std* - The standard GE grammar for the ant trails. 82

5.4 *adf* - GE grammar for the ant trails, with only one ADF. 82

5.5 adf_{dyn} - The grammar for the ant trails, which allows multiple function definition is shown below. `adf*()` is expanded to create unique signatures for the allowed functions. Then the function `adf*()` function call is used to determine which of the functions to call. 83
5.6 Symbolic Regression meta-grammar, *std* - For the standard GE grammar . 88
5.7 Symbolic Regression meta-grammar, *adf* - The GE grammar can define one function with one argument. 88
5.8 Symbolic Regression meta-grammar, adf_{dyn} - The GE grammar for creating any number of functions. Each function takes one argument. 88
5.9 Symbolic Regression meta-grammars, adf_{mg} - meta-GE grammar multiple parameters. 89
6.1 GEGA grammar for producing a bitstring of length eight. 96
6.2 GE bit string grammar with building block structures `<bbk4>`, `<bbk2>`. I.e. the reuse of groups of bits (building block structures) into a more compact representation of the bit string. 96
6.3 GE bit string grammar with choices between building block structures. Rules differing from Grammar 6.4 are shown in *italics* 97
6.4 Meta-grammar with building block structures example (mGGABB). 98
6.5 Example of a solution grammar produced by mGGABB (Grammar 6.4) . . 99
6.6 Meta-grammar with multiple building blocks (mGGAMBB). Rules differing from Grammar 6.4 are shown in *italics* 100
6.7 Example of a solution grammar from mGGAMBB (Grammar 6.6). In `<bit>` a bias towards `1` can be seen. 102
7.1 mGGAMBB example solution grammar from implicit sampling 119
7.2 mGGAMBB explicit sampling example solution grammar 120
8.1 Part of Grammar 6.6 which biases the grammar towards the use of building block structures. 129
8.2 mGGA grammar 1 130
8.3 mGGA grammar 2 130

8.4 GE bit string grammar with building block structures <bbk4>, <bbk2>. . . 131
9.1 Left-recursive grammar which can have an immediate change effect 154
9.2 Right-recursive grammar which can have an delayed change effect 154
10.1 Grammar for Ex. 14 165
10.2 Grammar variant for Ex. 14 166
10.3 Grammar for Ex. 15 168
10.4 Grammar for Ex. 17 175
11.1 Grammar bit string example, again 192
11.2 Simplified grammar in Grammar 11.1. The simplification affects the size of the derivation tree 198

Publications Arising

1. Erik Hemberg, Conor Gilligan, Michael O'Neill, and Anthony Brabazon. A grammatical genetic programming approach to modularity in genetic algorithms. In Marc Ebner, et al, eds, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 1–11, Valencia, Spain, 11 - 13 April 2007. Springer.

2. Jonatan Hugosson, Erik Hemberg, Anthony Brabazon, and Michael O'Neill. An investigation of the mutation operator using different representations in grammatical evolution. In *Proceedings of IMCSIT*, volume 2, pages 409–419, 2007.

3. Erik Hemberg, Michael O'Neill, and Anthony Brabazon. Grammatical bias and building blocks in meta-grammar grammatical evolution. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 3775–3782, 2008.

4. Erik Hemberg, Michael O'Neill, and Anthony Brabazon. Altering Search Rates of the Meta and Solution Grammars in the mGGA. *Lecture Notes in Computer Science*, 4971: 362, Naples, Italy, 26-28 March 2008, Springer

5. Erik Hemberg, Nicolas McPhee, Michael O'Neill, and Anthony Brabazon. Pre-,in and postfix grammars for symbolic regression in grammatical evolution. Workshop and Summer School on Evolutionary Computing Lecture Series by Pioneers, Derry, Northern Ireland, 18 -22 August 2008.

6. Michael O'Neill, Anthony Brabazon, and Erik Hemberg. Subtree deactivation control with grammatical genetic programming. In *IEEE Congress on Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence)*, pages 3768–3774, Hong Kong, 2008. IEEE Press.

7. Michael O'Neill, Erik Hemberg, Conor Gilligan, Eliott Bartley, James McDermott,

and Anthony Brabazon. GEVA:Grammatical Evolution in Java. *SIGEVOlution*, 3(2):17–23, Summer 2008.

8. Jonathan Byrne, Michael O'Neill, Erik Hemberg, Anthony Brabazon Analysis of constant creation techniques on the binomial-3 problem in grammatical evolution. In *Evolutionary Computation, 2009. CEC 2009*, Norway, 2009. IEEE Press.

9. Erik Hemberg An exploration of learning and grammars in grammatical evolution. In *Proceedings of the 11th annual conference companion on Genetic and evolutionary computation conference*, pages 2705–2708. ACM, 2009.

10. Jonatan Hugosson, Erik Hemberg, Anthony Brabazon, and Michael O'Neill. Genotype representations in grammatical evolution. *Applied Soft Computing*, pp.36-43 Vol.10 Issue.1, 2010.

11. O'Neill M., McDermott J., Swafford J.M., Byrne J., Hemberg E., Shotton E., McNally C., Brabazon A., Hemberg M. Evolutionary Design using Grammatical Evolution and Shape Grammars: Designing a Shelter. International Journal of Design Engineering, pp.4-24 Vol.3 No.1, 2010.

Chapter 1

Introduction

The research field of Natural Computation is where this thesis is set, a field which encompasses the study of concepts from nature and transforming them into problem solving tools for different environments. The goal of using computers to automatically solve problems is central to Artificial Intelligence and machine learning. By studying nature itself, we see its originality in how it devises varying solutions for a different range of environments by letting the fittest survive. Observing the resourcefulness of nature, especially as a computer scientist, a spontaneous reaction is to wonder how all this could be represented.

The focus of the investigation is on Evolutionary Computation, which is a parallel search of solutions, using grammars with a variable length representation, or, in more specific terms, the exploration of grammars in Grammatical Evolution. Grammatical Evolution (GE) is an evolutionary algorithm and the grammar is used to encode domain knowledge, the search itself is driven by evolution, hence Grammatical Evolution. The grammar in GE biases the search for solutions, and in combination with a variable length linear representation this is what distinguishes GE from other Evolutionary Algorithms (EA). The areas studied are the influence of the grammar used in the mapping process and the adaptation of the grammar during search. This is done both empirically and theoretically. After the experiments the role of the grammar in the search is further clarified along with a formal description of the GE algorithm and stricter definitions of the method used.

A grammar in Evolutionary Computation (EC) is an indirect form of representation, inducing bias to the search. Bias are all the factors that influence the form of each solution [160]. For a successful search, a proper representation of the problem and of the appropriate search operators is needed [137]. An improved understanding of the representation can help when trying to improve the performance of the algorithm used for search, e.g. when creating new operators as well as when distinguishing which applications the algorithm might suitably be used for. Hence, the representation is important for the efficiency of a search, for example a search using an indirect representation can be more efficient than one using a direct representation. This because each space has different compositions of properties, e.g. properties of constraints and neighbors. The contribution of this work is knowledge regarding the use of grammars in EC, more specifically, the use of grammars with the GE algorithm.

This chapter is structured as follows. In Section 1.1 the domain of Evolutionary Computation, as well as grammar representation will be introduced. In Section 1.2 the research aim, exploration of grammars in Grammatical Evolution, is introduced. The contributions are spelled out in Section 1.3. Limitations in the scope can be found in Section 1.4. Finally, the chapter ends with an overview of the proceeding chapters in Section 1.5.

1.1 Evolutionary Computation

EC adapts inspiration from nature by using a parallel trial-and-error approach, allowing the "fittest" to survive and reproduce. Organisms adapt by modifications, making them more fit for existence in their environment.

Background

In Fig. 1.1 the flow of a canonical EC algorithm is described. First initialize a population, then evaluate the population and give each individual a fitness, while the optimum is not found or max iterations are not reached: select individual solutions depending on their fitness from the population, apply operators to the selected solutions, which modify the

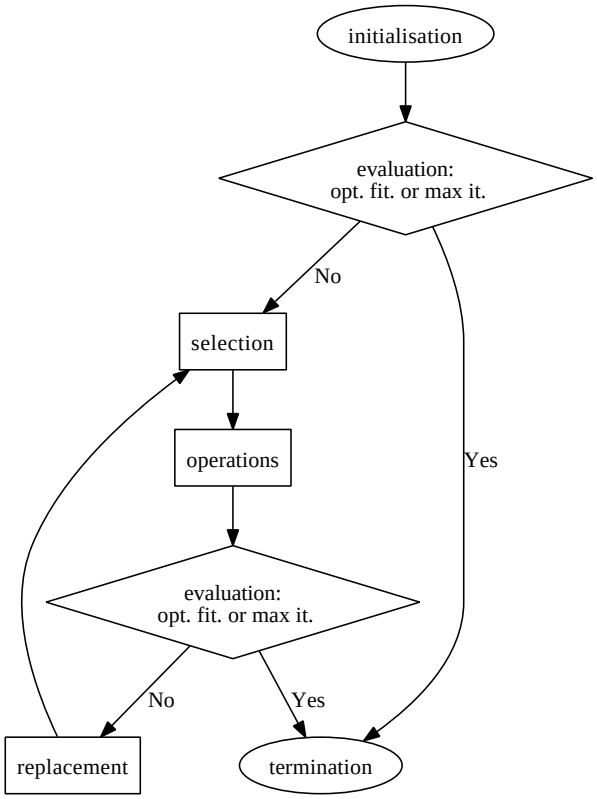


Fig. 1.1: Flow of a canonical Evolutionary Algorithm. First initialize a population, then evaluate the population, while not optimum found or not max iterations reached: select individual solutions from the population, apply operators to the selected solutions and replace the population.

solutions and allow the search to progress and replace the population given the individuals' fitness. Bäck et al. [7] give a more in-depth description of the different variations of EC.

In EC three big issues are: *a*) representation of the problem and potential solutions *b*) specification of the problem objective, *c*) issues involving the way that the search for a solution is conducted [165, 161]. Below follows a brief description of the different variants of EC, including how they differ.

Evolutionary Programming(EP) [38] Developed by Fogel in the 1960s. Finite state machines are used as predictors. The genotype is typically fixed-length character strings, and the operators are often mutation and crossover.

Evolutionary Strategies(ES) [10] Developed by Schwefel in the 1960s. The genotype is vectors of real numbers as well as representations of solutions. The genotype often includes self-adaptive mutation rates by adding a normally distributed random value to each component of the vector.

Genetic Algorithms(GA) [45] Developed by Holland in the 1970s. A fixed length binary genotype is used. Basic mutation is applied to each gene. One point crossover splits each parent at one point and exchanges the genes.

Genetic Programming(GP) [130, 86] Popularized by Koza in the 1990s. Individuals are variable length parse trees, executable code. GP is patented for LISP [82]. Both the structure and the contents of the solution are evolved.

EC methods have been applied to many different problems from optimization to simulation. To mention only a few applications: design en-route caching strategies [17], an approach for network coding [79] and EC for crystal structure prediction [105]. A large GP system for automated reverse engineering of nonlinear dynamical systems is presented by Bongard and Lipson [11]. Moreover, EC has also been used for evolving DNA-motifs [84, 49, 16].

The investigation in this work is the grammar-based GP methodology called GE, which is one of the subdivisions of GP. In GE a grammar representation is used to bias the search.

1.1.1 Grammars - Representation

An intuitive description of a grammar is that of a mechanism for producing sets of strings [53, 12]. The use of a grammar in this thesis is to rewrite or generate sentences. The problem of representation was referred to by Wagner [154] as "how to code a problem such that random variation and selection can lead to a solution?"

Antonisse [6] used a grammar-based genetic algorithm. Banzhaf [8] and Keller and Banzhaf [75] has binary strings as genotypes and program trees as phenotypes and uses a Context-Free Grammar for repairing programs during the mapping from input to output.

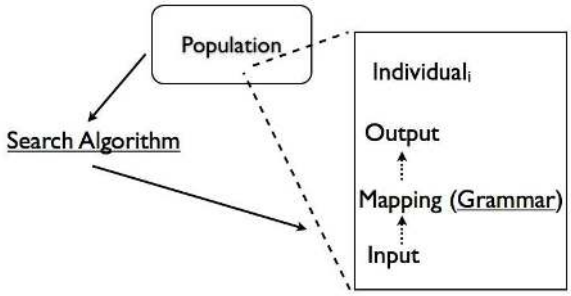


Fig. 1.2: The parts in GE which are explored are the operators in the search algorithm and the grammar used to map the input to the output.

This thesis examines grammars as a GP representation. Problems can be indirectly represented by a grammar. The use of a grammar constrains and biases the search space. The search space can be constrained by the declarative bias of the grammar, i.e. knowledge separate from a learning system [161].

Grammatical Evolution

It was O'Neill and Ryan [115] who introduced GE, a grammar-based form of GP. In GE the biological inspiration is the creation of protein from DNA and the feature distinguishing GE from other EAs is the use of the grammar and the redundant deterministic mapping used to generate output from input via the grammar. Already GE has been successfully applied to a wide range of problems [118], e.g. in finance [34]. Fig. 1.2 shows how the components of GE fit together and points at multiple parts where changes and learning can occur.

Not only the operators in the search algorithm influence the outcome, but when applying an algorithm that maps input to output via a grammar, the grammar itself also affects the result. In GE, the sequence of the input for the mapping to the solution can be learned using any search algorithm, e.g. GAs or Particle Swarm Optimization [107]. Standard GE uses standard GA mutation and single-point crossover.

Adaptation

Adaptation occurs after alterations or change. In order for the search to progress, either the solution or the problem is changed. One example of change could be the existence of noise, i.e. the change only occurs by random chance and is non-deterministic. When attempting to solve a problem a key question is what and how to change. When the EC trial-and-error, "survival of the fittest" heuristic is used the complication lies foremost in determining an appropriate fitness measure. In addition to the evolutionary component it is possible to add other algorithms or heuristics for learning. These components give the search bias and learning bias. Moreover, in order to explain adaptation we can study the relationship between modules and adaptation. It can be said that modules, depending on context, can both facilitate and complicate adaptation [39]. This ability depends on how abstractions are captured. Furthermore, it is argued that correct modules require less change for a successful adaptation.

1.2 Research Aims

In evolutionary search with a grammar, the grammar is important because it constrains and biases the search space by the probability of generating different sentences and by the possible sentences in the language given by the grammar. A grammar is an indirect encoding and in the case of GE it is generative, i.e. the input is rewritten to an output. Here, the role and importance of the grammar in GE will be explored. The long term goal is to be able to add to the understanding of how to automatically solve problems.

1.2.1 Questions

The aim is to understand *the role of grammars in Grammatical Evolution.*

Performance

1. *How does the grammar mapping affect the performance of GE?*

2. *Can the use of a meta-grammar improve the performance of GE when problems of a different scale are approached?*

3. *How does the grammar design influence the performance of GE?*

Adaptation

4. *Does the evolutionary learning of a grammar facilitate the capturing of modules?*

Theory

5. *How does the representation in GE react to changes in the input, i.e genotype?*

6. *How should different grammars be measured and compared?*

1.2.2 Objectives of Thesis

In order to comply with the research aims the following objectives have been achieved:

1. Survey the state of the art for grammars in EC and in GE.

2. Implement an open source software library for GE.

3. Identify and explore areas for exploration of grammars in GE regarding performance and adaptation.

4. Gain a wider understanding of GE by theoretically analyzing the results.

5. Draw conclusions from the empirical and theoretical results.

Method

The evolutionary computation method, Grammatical Evolution, was used to investigate the impact of the grammar representation. The experiments were run on different versions of GEVA [118] on a variety of Apple Macintosh computers with Intel Processors (Max 10 different machines, or 30 central processing units) manufactured from 2005-2009 with varying versions of OS X.

1.3 Contributions

A number of publications have come out of this work and are listed in the Publications Arising on page xviii. The exploration of grammars in GE has given rise to a number of both practical and theoretical contributions which are outlined here:

Literature review Review of literature in EC regarding algorithms that use grammars in Section 2.2, review of the use of grammars in GE in Section 3.1.4 and modularity in EC in Section 5.1.

GE software library Developed, implemented and released under an open source license Grammatical Evolution in Java (GEVA) [118], which has been used to implement experiments for several publications.

Grammar mapping In Chapter 4 when using grammars to examine how the mapping order, i.e. the expansion of non-terminals, is changed it was noted that the number of invalid individuals, i.e. individuals that do not map to a valid solution, was tied to the grammar and to the mapping order. Moreover, the results confirmed the expectation that the choice of grammar can produce performance advantage for the problems examined.

Explored meta-grammar & scalability The meta-grammar concept for GE, i.e. allowing a larger grammar with different bias, was applied to problems of increasing size in Chapter 6. The ability of the meta-grammar to scale to larger problems was confirmed.

Explored meta-grammar & modularity Modularity when using meta-grammars was explored both in a fixed-length solution context (Chapter 6, 7 & 8) as well as in a variable-length solution context in Chapter 5. The benefits from a representation which has bias towards modules, e.g. Automatically Defined Functions or building block structures in the grammar in the solution for benchmark problems were verified.

Introduced meta-grammar operators Operators for meta-grammars were introduced and examined in Chapter 7. It was confirmed that a slower rate of change (mutation) for the meta-chromosome can improve performance.

Explored meta-grammar grammars The meta-grammar grammars were investigated further, with respect to their bias in Chapter 8. The capability of the meta-grammar to use the building block structures provided in the grammar was shown. One recommendation arising from this study is to adopt a meta-grammar that allows the use of both a GA bit string representation in conjunction with the modular building block structures.

Formally described GE In Chapter 9 a formal description of GE is proposed and allows us to clearly show the different representations within the algorithm.

Theoretically analyzed the impact of change on GE input How an indirect representation from a linear input sequence reacts to changes was studied in Chapter 10. Different types of change considering change to input (genotype) were labeled and how these were propagated into other change types in the output (phenotype) via the linear mapping in the Context-Free Grammar. The conclusion is that the fewer non-terminals there are in the grammar, the less susceptible it will be to disruption. Furthermore, the effects of a change on the input were labeled.

Theoretically analyzed meta-grammar mapping The mapping process involving meta-grammars was explored in Section 10.5. The added dependence on the meta-chromosome for the solution chromosome and how the effects of change for a meta-grammar setup were also examined.

Introduced a GE schema In Section 10.4 a GE schema theory was introduced. Some operators were examined in relation to how sequences of the individual genotype are propagated over a generation. It showed that the canonical GE mutation is quite similar to crossover.

Explored grammar measurement in GE Chapter 11 investigated how static analysis can distinguish grammars in EC. A matrix representation for determining the length of the expansion of non-terminals in a grammar, the Expected Derivation Length (EDL) was introduced. Furthermore, a binary measurement of the convergence of a Probabilistic Context-Free Grammar in GE was presented.

The main conclusions are that the grammar biases the search and that it is possible to modify the grammar or operate on the structures created during the mapping to impose better bias for the search. We also know of different types of bias that will occur in grammars that are used. Furthermore, we can explore the different types of change that a grammar or an operator promotes, thereby allowing us a better comparison of our GE settings. The theory also presents new questions about how GE works and guide future research in answering them.

1.4 Limitations

The focus is on exploring the grammar in GE. In order to do this, the use of simple problems can improve the understanding of the grammar and thus obscuring the analyses of the algorithm by the complexity of the problem itself can be avoided. The grammars explored have been restricted to Context-Free Grammars.

When running an evolutionary algorithm, there are a number of direct and indirect choices to be considered, e.g. the choice of operators and parameter values. Furthermore, different problems have dissimilar behavior for various algorithms and settings. There was no exhaustive search of these settings here in any way. As for the grammar, neither has there been an exhaustive search of possible grammar combinations here. The foremost aim has been to inquire about the impact of the grammar in GE search, which will then allow optimization of the performance of the algorithm in the future.

Relevant Literature

In this thesis, the aim has been to study grammars in their EC context. This does not imply that there is no valuable knowledge to be gained from other fields. There are many good studies e.g. regarding schemas in both logic, mathematics and EC that have not been drawn upon. There is a great potential for finding additional metrics and analyzes for the grammars and schemas which might prove useful. There is also a vast literature on formal languages and computational linguistics regarding grammars.

1.5 Thesis Overview

The aim is to explore grammars in Grammatical Evolution. In order to understand the effect of grammars in GE both empirical and theoretical aspects are investigated. A broad and novel understanding of grammars in GE is made possible by these studies. For the purpose of producing a clear narrative the thesis is broken up in to four major parts. Each part contains several short chapters, this to clearly separate as well as to allow digestion of the different aspects of the research.

In Part I the preliminaries for the thesis are covered and the initial steps of our exploration are described. This part presents an overview of research areas for the empirical and theoretical investigations that will help us understand the grammars in GE and its impact on performance and adaptation. In Chapter 2, terms and concepts regarding grammars are defined. Previous work regarding grammars in EC is reviewed and gaps regarding the understanding of grammars and the representation of individuals are revealed. Chapter 3 describes the concept of GE and different variants are presented. Moreover, research opportunities, such as grammar order in mapping, meta-grammars and formal description are presented.

When attempting to understand grammars in GE it helps to first understand how they work in practice. The areas of grammar mapping in GE, modularity by using larger grammars, scalability and grammar design will be examined in Part II as well as the

questions of how the grammar can be used to improve performance and how it can be altered. This is explained in Chapter 4. Here, the understanding of the grammar and mapping in GE is examined by studying the mapping order, which gives insight into the impact of the derivation order. It also shows how the grammar input is related to the rule order bias.

The meta-grammar studies investigate how a larger grammar with a modified representation performs. Chapter 5 explores Automatically Defined Functions for meta-grammars. The studies of the meta-grammars are extended in Chapter 6 to investigate not only the ability to capture modules, but also the scalability. Moreover, the impact of operators on the meta-grammar implementation as well as grammar design are studied in Chapter 7. These reveal that the meta-grammar scales well for regular problems of increasing size. The effects of the meta-grammar setup reveal that a grammar design which includes equal bias to the use or non-use of building block structures has less variability in performance than a grammar with a strong bias towards building block structures.

The theory in Part III further investigates and tries to formalize and generalize the results from Part II. Gaps from GE are covered by the theory and questions raised by the experiments are addressed, e.g. how GE reacts to changes and different operations. A more rigorous description of mapping and the search spaces of the algorithm is given in Chapter 9 and in order to theoretically and more clearly understand the impact of grammars the entire mapping process is dissected. How a grammar affects the search is studied in Chapter 10 by analyzing the different types of changes that can occur from the use of a grammar in the mapping. This further clarifies the grammar's role in the mapping and the bias it gives to the search. Chapter 11 considers measurements of the grammars.

In Part IV conclusions are drawn. Chapter 12 contains these conclusions and an outline of future work.

Part I

Preliminaries - Preparing for an Exploration of Grammars in Grammatical Evolution

In Part I the preliminaries for the thesis are covered and the initial steps of our exploration are described. This part presents the map of research areas for the empirical and theoretical investigations that will help us understand the grammars in GE and their impact on performance and adaptation.

In Chapter 2 terms and concepts regarding grammars are defined. Previous work regarding grammars in EC is reviewed and gaps regarding the understanding of grammars and the representation of individuals are revealed. Chapter 3 describes GE and different variants are presented. Moreover, research opportunities, such as grammar order in mapping, meta-grammars and formal description are presented.

Chapter 2

Grammar Representation

The key parts in this chapter are the definitions of grammar and how other EC algorithms are using grammars and measurements in EC. This shows gaps in the understanding of how a grammar performs and adapts in evolutionary search.

Section 2.1 will introduce formal definitions of Context-Free Grammars (CFGs) and Probabilistic Context-Free Grammars (PCFGs) to aid the study of mapping in Grammatical Evolution (GE), these definitions will be used for the description of mapping and how output reacts to input changes, as well as for identification of grammar properties later in Chapter 3 and 9 and 11. Section 2.2 surveys grammars in evolutionary computation. The further complexities of grammars are examined in Section 2.2.2. Finally, Section 2.3 summarizes the chapter.

2.1 Grammar Definitions

Formal language theory deals with sets of strings which are called languages and with mechanisms for recognizing and generating them [53]. Consequently, here the grammar is considered in a computer science context for its syntactical properties. An intuitive description of a grammar is that of a mechanism for producing sets of strings [53, 12]. The use of a grammar is to rewrite or generate sentences.

Tab. 2.1: The Chomsky Hierarchy

Grammars	**Languages**	**Automata**
Phrase-structure Type 0 Context-sensitive with erasing	Recursively enumerable sets	Non-deterministic or deterministic Turing machines
Context-sensitive Monotonic	Context-sensitive	Non deterministic linearly space bounded Turing machines
Context-free	Context-free	Non-deterministic push down automata
$LR(k)$	Deterministic context-free	Deterministic push down au- tomata
Linear	Linear context-free	Two-tape non-deterministic fi- nite automata of a special type
Right linear Left linear	Regular sets	Non-deterministic or deterministic, one-way or two-way finite automata

The Chomsky hierarchy [53] for formal languages is shown in Tab. 2.1.

Finite state languages are a subset of context-free languages, the extra power of context-free languages is self-embedding or recursion, rewriting rules of the form $A \rightarrow \alpha A$, see Booth [12].

2.1.1 Context-Free Grammar

In a Context-Free Grammar the generation of a word is not dependent on the surroundings, see Booth [12].

Definition 1 (Context-Free Grammar (CFG)) A CFG is a four tuple $G = \langle N, \Sigma, R, S \rangle$, where:

- N is a finite non-empty set of non-terminal symbols.
- Σ is a finite non-empty set of terminal symbols and $N \cap \Sigma = \emptyset$, the empty set.
- R is a finite set of production rules of the form $R : N \mapsto V^* : A \mapsto \alpha$ or (A, α) where $A \in N$ and $\alpha \in V^*$. V^* is the set of all strings constructed from $N \cup \Sigma$ and $R \subseteq N \times V^*$, $R \neq \emptyset$.
- S is the start symbol, $S \in N$. □

```
<bitstring> ::=  <bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4>
<bbk4> ::= 1 1 0 0
         | 0 0 <bit> 1
<bit> ::= 1
```

Grammar 2.1: Example of a CFG in BNF for generating a bit string

"Context-Free" means that for a rule $A \to \alpha$, A can always be replaced by α, regardless of context [53]. A CFG can have many forms: an example of two different forms is a CFG with only one non-terminal $|N| = 1$ and Chomsky Normal Form (each rule leads to either two non-terminals or one terminal, $\forall A \in N, R : A \mapsto BC,\ A, B, C \in N,\ \alpha \in \Sigma$, making a derivation tree a binary tree which is valid if S does not generate an empty string).

The grammars in this thesis will be described using Backus-Naur Form (BNF), which is a meta-syntax to express CFGs in computer science. Knuth [80] describes the BNF as a set of production rules written as

```
<non-terminal> ::= expression
```

Non-terminals are enclosed between `<>`, alternatives for a definition are grouped together, `::=` separates left-hand from right-hand side and different production rules are separated by `|`. `expression` is a sequence of one or more symbols `expression` $\in V^+$.

Example 1 (CFG) A CFG grammar for generating a bit string can be written as $N = \{\texttt{<bitstring>}, \texttt{<bbk4>}, \texttt{<bit>}\}, \Sigma = \{\texttt{1}, \texttt{00}, \texttt{1100}\}$ and $S = \texttt{<bitstring>}$. Here Σ is defined to contain symbols which are the longest consecutive combinations of 1 and 0. Grammar 2.1 could also be written with $\Sigma = \{\texttt{0}, \texttt{1}\}$ the rules and the grammar in BNF.

A grammar generates a language $L(G)$, see Wetherell [157]. The following defintion of rewriting or generation is used, see Harrison [53]

Definition 2 (Generation) Let $G = \langle N, \Sigma, R, S\rangle$ be a context-free grammar and let $\alpha', \beta' \in V^*$. α' directly generates β', written as $\alpha' \Rightarrow \beta'$ if there exist $\alpha_1, \alpha_2, \alpha, \beta \in V^*$, such that $\alpha' = \alpha_1\alpha\alpha_2, \beta' = \alpha_1\beta\alpha_2$ and $\alpha \to \beta$ is in R. □

```
<bitstring> ::=  <bbk2><bbk2>
<bbk2> ::= 1 0
         | <bit> 1
<bit> ::= 1
```

Grammar 2.2: An example of a BNF for a bitstring of size 4 with three rules.

Note that the multiple-step generation, $\stackrel{*}{\Rightarrow}$ is the reflexive-transitive closure of $\Rightarrow$. A set is closed under some operation if application of that operation on members of the set always produces a member of the set. A set that is closed under an operation satisfies a closure property. From Harrison [53]

Definition 3 (Reflexive-Transitive closure) Let $\rho \subseteq X \times Y$ and $\sigma \subseteq Y \times Z$ be binary relations. The composition of ρ and σ is:

$$\rho\sigma = \{(x,z)|(x,y) \in \rho, (y,z) \in \sigma \text{for some} y \in Y\} \subseteq X \times Z.$$

For binary relations on a set, $\rho \subseteq X \times X$, the equality or diagonal relation is $\rho^0 = \{(x,x)|x \in X\}$. For each $i \geq 0$, $\rho^{i+1} = \rho^i\rho$. The reflexive-transitive closure of ρ is:

$$\rho^* = \bigcup_{i \geq 0} \rho^i$$ □

A *sentential form* of G is $S(G) = \{x : S \stackrel{*}{\Rightarrow} \alpha, \alpha \in V^*\}$. If $\alpha \in \Sigma$ then it is called a sentence.

Σ^* denotes the set of all finite length Σ sequences [53].

Definition 4 (Language) The language generated by G is $L(G) = S(G) \cap \Sigma* = \{x : S \stackrel{*}{\Rightarrow} x, x \in \Sigma^*\}$. □

Example 2 (Sentence generation/derivation) From Grammar 2.2 we get

$$\begin{aligned} \texttt{<bitstring>} &\Rightarrow \texttt{<bbk2><bbk2>} \\ \texttt{<bbk2><bbk2>} &\Rightarrow \underline{\texttt{10}}\texttt{<bbk2>} \\ \ldots &\Rightarrow \texttt{1011} \\ \texttt{<bitstring>} &\overset{*}{\Rightarrow} \texttt{1011} \end{aligned}$$

Note that an underlined word indicates the rewritten symbol. □

This section has provided definitions of Context-Free Grammars, language- and sentence generation. Now we will introduce Probabilistic Context-Free Grammars, which will allow us to analyze different grammars used in GE in Chapter 11.

2.1.2 Probabilistic Context-Free Grammar

The CFG can be expanded to a Probabilistic Context-Free Grammar, where each rule has an associated probability, see Wetherell [157].

Definition 5 (Probabilistic Context-Free Grammar (PCFG)) The Probabilistic CFG is the tuple $\langle G, P \rangle$, where G is a CFG and P is an ordered set of probabilities $\{p_{ij}\}$,i is the index for the non-terminal left-hand side and j is the index for the productions with the same left-hand side.

- For all $r_{ij} \in R$ there exists one probability $p_{ij} \in P$. (for r_{ij} i means left-hand side(LHS) number i in the BNF and j means production number j on the right-hand side for LHS i, see 3)
- For each $p_{ij} \in P, 0 \leq p_{ij} \leq 1$. If $p_{ij} = 0$ then r_{ij} can be eliminated from the grammar.
- For all $r_{i*} \subseteq R$, i.e. $*$ is a wildcard and r_i is the restriction $R\Big|_{\{n_i\}}, \{n_i\} \in N$, $\sum_{0 \leq j \leq |r_{i*}|} p_{ij} = 1$. $|r_{i*}|$ is the number of productions with the same non-terminal

left-hand side. Let $n_i \in N$ be the non-terminal with index i; then

$$r_{i*} = \{(n_i, r(n_i)) : r \in R\} \subseteq R \tag{2.1}$$

□

Example 3 (PCFG) The PCFG grammar in Backus-Naur Form is Grammar 2.1

$$r_{00}, p_{00} = P_0 = 1, \texttt{<bitstring>} \longrightarrow \texttt{<bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4>}$$
$$r_{10}, p_{10} = P_1 = 0.5, \texttt{<bbk4>} \longrightarrow \texttt{1100}$$
$$r_{11}, p_{11} = P_2 = 0.5, \texttt{<bbk4>} \longrightarrow \texttt{00<bit>1}$$
$$r_{20}, p_{20} = P_3 = 1, \texttt{<bit>} \longrightarrow \texttt{1}$$

□

Now it is possible to look at derivations from the PCFG, where expansion of the grammar generates sentences in the language. That is, first the start symbol is expanded, and then each non-terminal, to create a sentential form. The derivation is finished when there are only terminal symbols in the string, a sentence in the language, from Wetherell [157].

Definition 6 (Derivation) A derivation Δ in a grammar G is a sequence of production numbers, $\langle i_0, \ldots i_n \rangle$, such that

- For $0 \leq k \leq n$, P_{i_k} is a production of G.
- For each k there exists a sentential form $\delta_k = \alpha_k A_k \beta_k,\ A_k \in R,\ \alpha_k, \beta_k \in V^*$.
- There is a string $\delta_{n+1} \in V^*$.
- $\delta_0 = S$ (i.e. $\alpha_1 = \beta_1 = \Lambda$)
- For each $0 \leq k \leq n, P_{i_k} = A_k \rightarrow \gamma_k$.
- For each $0 \leq k \leq nk, \delta_k = \alpha_k A_k \beta_k \Rightarrow \alpha_k \gamma_k \beta_k = \delta_{k+1}$.

□

From Def. 6 it is possible to give a probability to a word, see Wetherell [157].

Definition 7 (Word probability) The probability of a word $w \in \Sigma^*$ is

$$p(w) = \prod_{1 \leq k \leq |\Delta|} p_k \tag{2.2}$$

where $|\Delta|$ is the length of the derivation. □

Example 4 (Word probability) Using Grammar 2.1 the word `11001100110011001100110011001100` has the probability $p(w) = (1/2)^8$ (note that in Grammar 2.1 all words have the same probability) □

Here, we define a derivation tree as multiple derivation steps, as Whigham [158].

Definition 8 (Derivation tree) The derivation tree from the start symbol is denoted by $D := \{S \stackrel{*}{\Rightarrow} \alpha, \alpha \in \Sigma^*\}$. □

In the derivation tree a branch is defined as

Definition 9 (Branch) The branch is denoted $D(A)$, from the non-terminal A, therefore $D(A) = \{A \stackrel{*}{\Rightarrow} \alpha, \alpha \in \Sigma^*, A \in N\}, D(A) \subset D$ □

Within the input sequence and derivation tree there are partial derivation trees called subtrees, Whigham [159].

Definition 10 (Derivation subtree) A subtree is $D_N = \{x \stackrel{*}{\Rightarrow} \alpha, x \in N, \alpha \in V^*\}$. □

Differences in derivations are distinguished by their derivation trees, see Wetherell [157].

Definition 11 (Derivation difference) Two derivations, δ and δ' are different if their derivation trees are different. □

To summarize, this section has given definitions of CFGs, PCFGs and derivations, all needed for future descriptions of mapping and representation.

2.2 Grammars in Evolutionary Computation

This section first recounts previous use of grammars in EC and then surveys some grammar-based algorithms and identifies research opportunities. An early paper on work using Evolutionary Computation (EC) and a grammar formalism was published by Hicklin [63]. For GAs Antonisse [6] used a grammar-based genetic algorithm. Also Johnson and Feyock [68] use a grammar to acquire expert system-rule bases, their algorithm generates LISP-like rules for expert systems by using variable length integer strings and a CFG. This system is quite similar to GE, but does not use the same mapping function and chromosome representation, which leads to a different implementation of the operators. Another example of early work with EC and a grammar is the one done by Gero et al. [43].

Grammars constrain and bias the search [130], i.e. the indirect encoding of the grammar allows search space transformations. For each grammar there is a distribution of output strings. The hope is that the grammar transforms the fitness landscape to make it easier to solve and that the real solution is in the language the grammar creates. The possible variations of a grammar allow for a large number of different grammars. Moreover, even some problem structures are grammar-related, e.g. the task of finding regular expressions. One observation regarding the grammar encoding is that it can sometimes make the causality constraint in the mapping from input to output difficult to follow.

A grammar can be static or dynamic and the grammars restrict the search space, McKay et al. [93]. This can reduce the cost to find the solution, but in the worst case it might also restrict the search to a space that does not contain a solution, or it might make the space difficult to search. A common approach is to start with a general grammar interactively modified by the user between runs.

Kargupta and Ghosh [69] investigate genetic code-like transformations for machine learning, showing that genetic code-like transformations can construct a representation that makes the learning problem easier to solve. Toussaint [151] discusses mapping in EC and, focusing on variability at the evolution of output, proposes a theoretical framework for evolution of complex input-output mappings. Keller and Poli [76] use grammars in a

cost-benefit investigation of a linear GP hyper heuristic.

Poli and McPhee [129] use an n-gram system to implement linear GP. Given a language defined over some set of symbols, an n-gram is an ordered list of n symbols. For example, in the English sentence "The cat sat on the mat", "the cat" and "cat sat" are 2-grams, while "the cat sat" is a 3-gram. An n-gram model is a type of probabilistic language model based on learning the probabilities of possible n-grams from a language source. Later, the developmental plasticity of linear GP is explored to show evolution of programs with reuse and variation of the instruction sets used in the solutions [96].

This section has listed work with grammars in EC and presented examples of work where changing the representation can improve the performance of the algorithm. When using grammars it is important to get the grammar bias right, to ensure that the search space covers the optimal solution. Moreover, there should be a high probability of finding solutions that lead to the optimal solution in the language that the grammar generates. There are still gaps in our knowledge of how grammars work in EC.

2.2.1 Grammar-based Algorithms

In Genetic Programming one issue with the representation is the closure of the expression, in canonical GP only one type is allowed. This issue is addressed by Strongly Typed GP, see Montana [98]. Typing can be handled by grammars as well. Whigham [158] added grammars to the derivation trees for a more expressive syntax. Paterson and Livesey [125] evolve caching algorithms in C and uses a fixed genotype to encode the indices for derivation rules in a grammar.

In a study of grammars and evolutionary learning Whigham [161] desires a machine that learns how to construct relationships based on a representation of the problem, and considers learning as the search for one particular object from a large set of possible objects. Learning with grammars in is done by updating the probability of rules by the frequency of rule use in superior individuals. Also, new production rules can be added, these are learned from superior individuals and chosen to have minimum impact on the original grammar.

Grammars can be used in conjunction with EAs to create Estimation of Distribution Algorithm Genetic Programming (EDA-GP) [144], where the probabilities and/or the structure of the grammars are changed. The aim is to infer a grammar that captures the correct model and where the grammar is a possible representation for constraints and bias. There are different approaches of how to infer models, e.g. Minimum Description Length. One common denominator is that it is computationally expensive to infer the model.

Learning can further be broken into levels and modules, where the modules are investigated or the links between them or even both. An overview of some representative algorithms with adopting grammars is presented in Tab. 2.2. In the remainder of this section they will be elaborated on.

The algorithms are divided by the structure of the individual solutions, that is, the structure of the individuals is either tree-based, or linear. If it is an EDA the strucure is non applicable (NA). The structure of the individual affects many of the EA operators used. For EDAs the exploration of the search space is performed by adapting and sampling probability distributions instead of using traditional genetic operators [130].

Tree-based

Algorithms with a tree-based representation and which use a grammar.

GGP Whigham [161] uses a grammar to constrain the individuals to only grammatically correct individuals, thus adding domain knowledge and biasing the search. The population is based on derivation trees on which the evolutionary operators operate.

DCTGP Logic-based GP with Definite Clause Translation Grammar Ross [136]. A DCTG is a logical version of an attribute grammar, this permits the grammar-based GP system to define non-trivial semantics. It has been used for the evaluation of a stochastic regular motif language for protein sequences [135].

TAG-GP Hoai and McKay [64] investigate grammar- guided genetic programming with Tree-Adjunct grammars (TAG). Hoai et al. [65] investigate representation and struc-

Tab. 2.2: Evolutionary Algorithms that explicitly use a grammar and learn. The columns show if the parts are static (S) or dynamic (D). Representation describes if a specific type of grammar is used. Structure refers to how an individual is represented. The algorithms are divided by the structure of the individual solutions, that is, the structure of the individuals is either tree-based, or linear. EDA is implied with NA.

Algorithm	**Mapping**	**Representation**	**Learning**	**Structure**
Grammatical GP(GGP), Whigham [161]	S/D	CFG	S/D	Tree
Genetic Algorithm for Deriving Software (GADS), Paterson and Livesey [126]	S	CFG	S	Linear
Definite Clause Translation Grammars GP (DCTG-GP), Ross [136]	S	DCTG	D	Tree
Developmental GP(DGP), Keller and Banzhaf [77]	D	CFG	D	Linear
Stochastic Context-Free Grammar GP (SCFG-GP), Ratle and Sebag [131]	D	PCFG	D	Tree
TAG-GP Hoai and McKay [64]	S	TAG	S	Tree
EDA-GP, Bosman and De Jong [13]	D	CFG	D	NA
PEEL, Shan [143]	D	PDF	D	NA (Linear)
Chemical GP, Piaseczny et al. [127]	S	Grammar	D	Linear
AGBGP, Wong [167]	D	Logic Grammar	D	Tree
Bayesian Automatic programming(BAP), Regolin and Pozo [133]	D	CFG	D	NA
Probabilistic Adaptive Mapping(PAM GP), Wilson [165]	D	CFG	D	Linear
LPCSG, Tanev [150]	D	CSG	D	Tree
Shared Grammar Evolution, Luerssen and Powers [90]	D	CFG	D	Linear
PCFG-LA, Hasegawa and Iba [54]	D	PCFG	D	NA

tural difficulty in Genetic Programming using TAG and show that with this representation and using simple insertion and deletion operations very difficult GP problems become easy to solve.

SCFG-GP Ratle and Sebag [131] introduced Stochastic Context-Free Grammar GP (SCFG-GP), a technique in which programs are automatically created. They applied a stochastic generative grammar together with a method of updating the grammar's probabilities based on the productions used in the best programs in previous generations. There was no transmission of genetic material via crossover. In the simple ("scalar") version of this technique, a single vector stored the grammar's probabilities. A more sophisticated ("vectorial") version maintained one vector per possible depth in the derivation tree, so that production probabilities were depth-dependent. The motivation for depth-dependence was to allow some productions (e.g. recursive ones) to be more likely early in the derivation, and others (e.g. non-recursive ones) more likely at higher depths.

Adaptive Grammar Based Genetic Problem Wong [167] has a flexible framework called GGP (Generic Genetic Programming). To learn programs in different languages the framework combines GP and Inductive Logic Programming, the use of mathematical logic as a representation for examples, background knowledge and hypotheses. The system can represent context-sensitive information and domain-dependent knowledge. GGP is based on logic grammars because they are more expressive than CFGs in representing context-sensitive information and domain knowledge for the induced target program. An extended-logic grammar differs from a CFG in that the grammar symbols, may include arguments.

LPCSG Tanev [150] applies Learning Probabilistic Context-Sensitive Grammar (LPCSG) which uses a table for probability distributions for rules with multiple productions, with probabilities for each context and these are updated during evolution. Both context of rule and probability are learned.

Linear

Algorithms with a linear representation and which use a grammar.

GADS Paterson and Livesey [126] introduced GADS, a technique for GP. The GADS genotype is a list of integers representing productions in a syntax. This is used to generate the phenotype. If the gene value is not within range of the number of production choices to be expanded it is skipped and a new gene is read.

DGP Developmental Genetic Programming (DGP) includes methodologies that explicitly set out to separate the genotype space from the phenotype (or solution) space through a connection (or mapping) between the two spaces, see Keller and Banzhaf [77]. First Keller and Banzhaf [77] co-evolves genotype and solution identical to O'Neill and Ryan [114], then Margetts and Jones [91] expands the study with adaptive DGP.

PAM GP separates mapping from genotype (they are united in the implementation of Keller and Banzhaf [77]). Mappings and genotypes are separated into two populations that co-evolve, see Wilson [165]. Wilson's system uses a mapping that can be seen as a table relating genotype segments (binary sequence codons) to symbol members of a function set [165, 164].

GE O'Neill and Ryan [115] The canonical form is inspired by the transcription and translation of a sequence of DNA into a protein. This is modeled by use of the grammar and the redundant deterministic mapping is used to generate output from input via the grammar. See Chapter 3 for further description of GE.

Shared Grammar Evolution Luerssen and Powers [90]. Luerssen [89] combines grammatical development with grammars in GP to establish declarative bias. Programs are generated by a global Context-Free Grammar that is transformed and extended by a user-defined grammar. Grammatical productions and encapsulated sub-routines are shared between programs. This allows reuse and reduces evaluations.

Chemical GP Suzuki et al. [149], Piaseczny et al. [127], allow feedback to the rewriting of rules in input-output mapping.

Estimation of Distribution Algorithms

Algorithms that estimate a distribution from which individuals are generated.

PCFG-LA Hasegawa and Iba [54] introduce a latent variable model for EDAs, i.e. a model where a variable is inferred from other observed variables. This model is based on a PCFG using different estimation methods. Also, PMBGPs (Probabilistic Model Building GP) with SCFG has been examined by Hasegawa and Iba [55, 56].

Bayesian Automatic Programming Regolin and Pozo [133] combine grammar evolution and stochastic models to evolve programs, using a Bayesian network to consider relations among production rules.

EDA-GP Bosman and De Jong [13] use a specific EDA for GP with a probabilistic model that employs transformations of production rules in a Context-Free Grammar to represent local structures. They infer grammar and structure, MDL is used to measure "goodness" of a grammar.

PEEL Shan [143], Program Evolution with Explicit Learning(PEEL) is a method which is to GP as estimation of distribution algorithms is to GAs. It represents knowledge explicitly, by using a table that describes the search space, which is incrementally built, instead of using an implicit representation by a population. The table consists of rules describing the likelihood that a given production will result from a given non-terminal, under some conditions of depth and location in the tree. These rules are added, refined, and updated according to the best individuals generated at each step. It does not use a true, ongoing population, traditional mutation or crossover operators.

This section has given some background to grammars in EC as well as a brief summary of some grammar-based EC algorithms. It has shown that grammars are used in many

different algorithms as well as ways in EC. The structures representing the individuals and the grammar are varying, all give different grammatical- and search bias and are adapted in different ways.

2.2.2 Measuring Grammars

In this section we consider the properties of a grammar and how to measure grammar complexity. The data that can be gathered from measuring a grammar can be used to address broader performance issues and measurements in EC. This leads to a compact introduction of performance measures in EC.

2.2.3 Different Grammar Measures

By using a grammar, the diversity of the solutions is affected. A grammar can be seen as a distribution, Poli et al. [130], where each word in the language is an event. Other modeling approaches of distributions can be given broadly by the EDA category. One difference of the Grammatical GP approach to that of EDAs is that it uses an explicit population. Shan [143] uses minimum description length (MDL) to learn a grammar, it is difficult to infer PCFGs. A grammar creates a bias towards a certain kind of connections. This will make the search successful if a solution has the same properties.

Lehman and Shelat [87], when discussing approximation algorithms for grammar-based compression, measure grammar complexity as the total number of symbols on the right-hand side of all rules. This measure is different to measuring the total size of the grammar. It might not be trivial to calculate the values for the measures exactly. Compression has been used as a measurement, Shin et al. [145] analyzed the regularity of GP genomes (trees) by using compression and expression simplification. This was further extended by McKay et al. [94] who used compression to understand how building blocks were distributed in GP populations. Other metrics for bounding probabilities could be useful, see Gibbs and Su [44].

The information gathered from measurement of grammars is helpful for guiding the

performance and the measure of the EA. In the next section some important performance measures will be mentioned.

2.2.4 Grammar Properties for Search

Whigham [160] names two important components of search bias, strength and correctness. A strong bias focuses on a small number of solutions and a weak bias has a larger number of solutions instead. Correctness describes the suitability of the bias to the problem, an incorrect bias does not allow the correct solution to be expressed. This leads to the problem of constraining the search space while not disallowing correct solutions.

Minimum Encoding Inference

Shan [143] compares PCFGs using a minimum encoding inference metric, Minimum Message Length (MML) or Minimum Description Length (MDL) [27]. Simpler models often generalize better on unseen data, and MML trades model complexity for goodness of fit. A model is worth considering if the shortening that the encoded data string gives is lower than the cost of representing the structure and parameters of the model. This balances the complexity and accuracy of the model. Thus, the model should minimize the cost of coding the data, $L(D)$ is the sum of the cost of coding the model $L(G)$ and the cost of coding the data given the model $L(D|G)$ yields $L(D) = L(G) + L(D|G)$. The probability of generating the solutions from a grammar is

$$p_D = -\log \prod_i^D p_i = -\sum_i^D \log p_i$$

To encode a PCFG the names of the terminals, the number of terminals, and the number of non-terminals need to be encoded. Moreover, for each rule the LHS, the right-hand side (RHS) and the probability need to be encoded. The probabilities are encoded by a symmetric Dirichlet prior. The combination of these gives the total cost of encoding the grammar.

The method described by Shan [143] is to infer a grammar. For GE this approach could be used to modify solution grammars with more than one sample. We are foremost interested in comparing the grammars without knowing how they fit the data. How the grammar fits the data can be added as an extra indicator. In addition, the MDL can be used as the objective for the fitness function, with the model complexity being added to the function, e.g.

$$h = L(G_s) + L(D|G_s)$$

where $L(D|G_s)$ is the currently used fitness function. Another approach would be to use it multi-objectively.

The straightforward approach is to use the cost function for a grammar developed by Shan [143]. It might be more sensible to introduce a meta-grammar GE specific cost function for the grammar, which takes into account the properties discussed.

Given the properties desired by a grammar a cost function can be devised. An example of a naive grammar cost function is:

- Number of terminals, $|\Sigma|$
- Number of non-terminals, $|N|$
- Number of rules, $|R|$
- Number of non-terminals in the rules, i.e. the total number of non-terminals on the right-hand side of the grammar
- Number of terminals in the rules, i.e. the total number of terminals on the right-hand side of the grammar
- MML of the probabilities

$$MML(\hat{\theta}, D_n, \alpha_i) = -\sum_{i=1}^{C} \log \hat{\theta}_i^{\alpha_i + n_i - 1/2} + \log B_C(\alpha_1, \ldots, \alpha_C) + 1/2(C-1)\log n + C/2(1 + \log 1/12)$$

One issue to consider is that in order to avoid premature convergence of the search a minimum model might not be the most efficient. Therefore a desired model is a combination of complex- enough models. Considering that some of the experiments have been dealing with changing fitness landscapes it might also change the desired complexity of the model.

This section has briefly reviewed how to measure grammars in EC.

2.3 Summary

The central contents of this chapter were the definitions of grammar and how other EC algorithms are using grammars and measurements in EC. Context-Free Grammars and derivations have been defined. This is needed for the exploration of grammars in Grammatical Evolution as well as for understanding of what a grammar is. The use of grammars in Genetic Programming has also been surveyed, along with the structures used to implement them as well as their approach to adaptation. Different measures of a grammar can be useful in EC, both simple and practical ones, e.g. number of rules in the grammar, as well as more theoretical measurements such as entropy, which are more difficult to compute. There are still gaps in the knowledge of how a grammar performs and adapts in evolutionary search. E.g. how grammars with large search spaces behave when only evolution is used to guide the search.

Now that Evolutionary Algorithms and grammars have been presented we move on to describe the Evolutionary Algorithm called Grammatical Evolution in more detail in Chapter 3.

Chapter 3

Description of Grammatical Evolution

Chapter 2 defined grammars, surveyed grammar-based algorithms in EC and measurements of grammars in EC. In order to complete the prerequisites for exploration of grammars in grammatical evolution, we now turn to a more in-depth description of the foremost studied algorithm in this thesis, Grammatical Evolution (GE). This chapter presents a broad overview of GE and of previous work and its components. GE is inspired by the transcription and translation of a sequence of DNA into a protein. This is modeled by use of the grammar and the redundant deterministic mapping used to generate output from input via the grammar. The research opportunities revealed in this chapter are the possibilities to further investigate grammar and search bias in GE and how they affect performance, as well as how grammars with large search spaces adapt. In addition, the chapter presents an occasion to more formally describe the algorithm and analyze how the grammar is affected by changes in the input.

The GE system overview is shown in Fig. 3.1. The grammar maps the input (genotype) to the output (phenotype). The phenotype is evaluated and the search is performed by operations that use the fitness values for selection and replacement, as well as modifying the genotype.

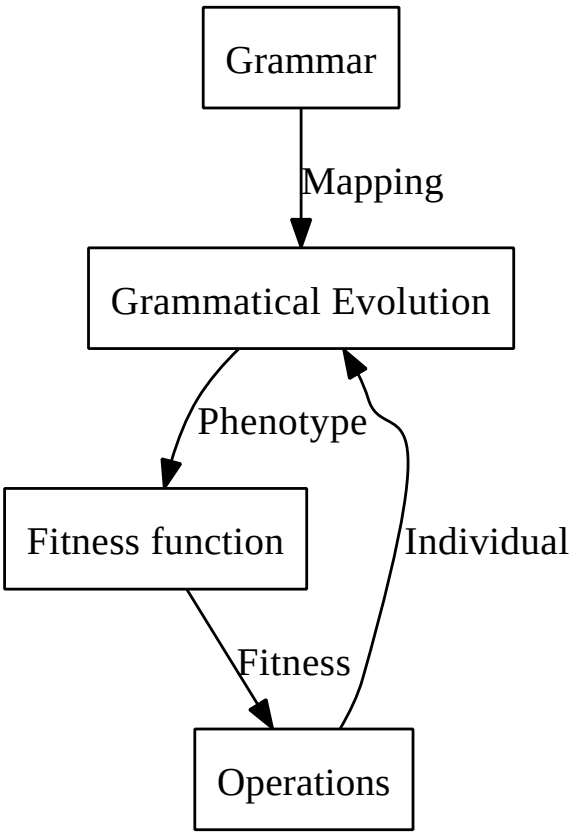


Fig. 3.1: Grammatical Evolution components. GE takes an individual where the grammar maps the input (genotype) to the output (phenotype). The phenotype is evaluated and the search is performed by the operations that use the fitness values for selection and replacement, as well as modifying the individual's genotype.

Section 3.1 delves into grammar representation, the mapping, and previous research in mapping. In Section 3.2 the operators used in GE are examined. Application areas of GE are presented in Section 3.3, before the chapter is summarized in Section 3.4.

3.1 GE Algorithm

Grammatical Evolution (GE) is a grammar-based form of GP. It is inspired by representation in molecular biology and combines this with formal grammars. The GE system is flexible and allows the use of alternative search strategies, whether evolutionary, deterministic or of some other approach. This system also includes the ability to bias the search by

changing the grammar used. Since a grammar is used to describe the structures that are generated by GE, editing the grammar modifies the output structures. This constraining power is one of GE's main features. The genotype-phenotype, i.e. input-output mapping means that GE allows search operators to be performed on any representation in the algorithm, e.g. on the genotype (integer or binary chromosomes), as well as on partially generated phenotypes, and on the completely generated derivation trees or phenotypes.

This section describes the inspiration for GE and the control flow of the algorithm and the different steps.

3.1.1 Biological Inspiration

The biological inspiration for GE comes from the generation of a protein from a sequence of DNA, which contains several mappings. A simplified description of the generation of a protein from DNA is described in Tab. 3.1. In Biology, the genotype, DNA is transcribed to RNA, the RNA is translated to amino acids, the amino acids create proteins, and the proteins generate a phenotype. Analogously, for an individual in GE the genotype, binary string, is transcribed to an integer sequence, the integers are translated to production choices via a grammar, and the phenotype is the sentence generated from the grammar.

3.1.2 GE Control Flow

In GE the control flow of an EA in Fig. 1.1 is extended with a genotype-phenotype mapping, this is the same as "decoding" in a GA. The canonical GE uses a standard GA as a search engine, with crossover and mutation. The steps in a single iteration of GE are generally:

1. **Initialization** Input in the initial solutions is generated, e.g. uniformly randomly generated integer sequences (see Section 3.1.2).

2. **Mapping** Mapping via a grammar, e.g. CFG (see Section 3.1.3).

   (a) **Binary to Integer** (Transcription) Binary to integer translation

Tab. 3.1: Comparison of a generation of a protein and the derivation of a sentence in GE. In Biology the genotype, DNA is transcribed to RNA, the RNA is translated to amino acids, the amino acids create proteins, and the proteins generate a phenotype. Analogously, for an individual in GE the genotype, binary string, is transcribed to an integer sequence, the integers are translated to production choices via a grammar, and the phenotype is the sentence generated from the grammar.

Biology		**Grammatical Evolution**
DNA		Binary string
⇓	*Transcription*	⇓
RNA		Integer sequence
⇓	*Translation*	⇓
Amino Acid		Production choice
⇓		⇓
Protein		Sentence(Program)
⇓		⇓
Phenotypic effect		Evaluated sentence

   (b) **Integer to String** (Translation) Grammar maps integer value to a sentential form (sequence of symbols).

3. **Evaluation** The individual solutions are evaluated.

4. **Operators** Operations on input, e.g. mutation and crossover (see Section 3.2).

   (a) **Selection** Some individuals from the current population are included in a new population (see Section 3.2.2).

   (b) **Variation operators** Individuals are modified by some operators, e.g. crossover and mutation (see Section 3.2.1).

   (c) **Replacement** A new population is created from the selected population and from the current population (see Section 3.2.2).

5. **Termination** When the start symbol has generated a sentence, the genotype (input) is extended by wrapping (see Section 3.1.2).

These steps complete the algorithm.

Initialization

The search must start with some initial solutions which later will be modified. For initialization the input is often uniformly randomly generated. The Ramped Half and Half Initialization[1] of GP has also been studied [115]; in order to increase the diversity of solutions in the initial population there is a start depth parameter and a maximum depth parameter. The current max initialisation depth increases from the minimum to the maximum depth in order to ramp up the depths of the solutions in the population. Two tree creation methods are combined, each with a 50% probability of being selected. The Grow tree generation randomly chooses a rule until the current max depth is reached. When using the Full tree generation a rule that will create a tree with the current maximum depth will always be chosen.

Termination - Wrapping

During the genotype-to-phenotype mapping process it is possible to use all codons in the genotype, and in this case the *wrap* operator is applied. This results in returning the the start of the genotype and reading the first codon in the individual, i.e. codons are re-used when wrapping occurs. GE works with or without wrapping, and wrapping has been shown to be useful for some problems [120]. However, with wrapping, an additional functional dependency between codons is introduced. Wrapping was further investigated [142] and a heuristic to minimize the number of wraps needed before the system can determine failure was presented. Hugosson et al. [67] investigated a novel wrapping operator for binary and Gray code representations, and found that across the problems examined there was no general trend to recommend the adoption of an alternative wrapping operator.

An individual that is not completely mapped, even after wrapping, is called an invalid individual. The number of invalid individuals can be reduced e.g. by strong selection pressure or by using steady-state replacement. Alternatively, a repair strategy which aims to make invalid individuals valid can be used [122], where only terminating rules are allowed

[1]Sometimes this is called Sensible Initialization in GE

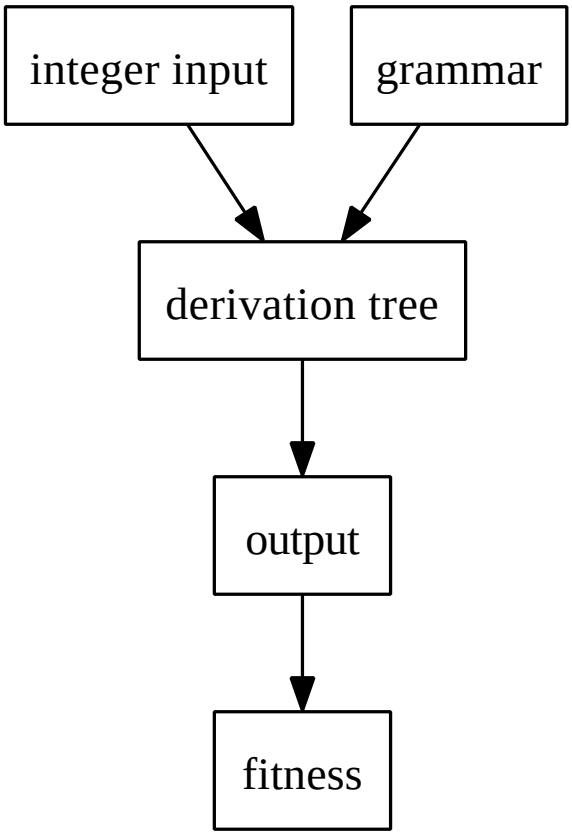


Fig. 3.2: GE mapping flow: input and grammar are mapped to output that is evaluated and assigned a fitness

to be chosen when all the codons have been used.

3.1.3 Grammar Mapping in GE

The mapping of GE is shown in Fig. 3.2. There are different spaces, genotype, phenotype and fitness.

The Grammar

For GE a suitable BNF grammar definition must exist. How much domain knowledge to incorporate is decided by the practitioner, who also defines how general or specific the Backus Naur Form (BNF) grammar is.

In GE, a BNF-grammar describes the output sentences that can be produced by the

```
<expr> ::= ( <expr> <biop> <expr> )
         | <uop> <expr>
         | <bool>
<biop> ::= and
         | or
         | xor
         | nand
<uop> ::= not
<bool> ::= true
         | false
```

Grammar 3.1: Example of a grammar for boolean expressions. `<expr>` has three production choices, `<biop>` has four production choices, `<uop>` has one production choice and `<bool>` has two production choices.

system, as well as the grammar bias.

Example 5 (Boolean grammar) The Grammar 3.1 can be used to generate boolean expressions, and `<expr>` can be transformed into one of three rules. It can become either `( <expr> <biop> <expr> )`, `<uop> <expr>`, or `<bool>`. From Definition 1 a grammar can be represented by the tuple $\langle N, \Sigma, R, S \rangle$.

N = { `<expr>, <biop>, <uop>, <bool>` }

Σ = { `and, or, xor, nand, not, true, false, (, )` }

S = { `<expr>` }

□

The code produced after mapping a BNF-grammar in GE will consist of elements of the terminal set Σ. The grammar is used in a generative approach, whereby the evolutionary process evolves the production rules to be applied at each stage of a derivation process (see Def. 6 on page 20), starting from the start symbol, until a complete program is formed. The mapping (derivation) is complete when the sentence is one that is comprised of only elements of Σ.

The Mapping

The genotype is used to map the start symbol into a sentence, by the BNF-grammar. The mapping is done by reading input(*codons*) to generate a corresponding integer value, from which an appropriate production rule is selected by using the following mapping function:

$$Rule = c \ mod \ r \tag{3.1}$$

where c is the codon integer value, and r is the number of rule choices for the current non-terminal symbol.

Example 6 (Choosing a production from a rule) Consider the following rule from the grammar in Grammar 3.1. Given the non-terminal `<biop>`, which describes the set of boolean operators that can be used, there are four production rules to select from. The choices are labeled from zero.

```
<biop> ::= and          (0)
         | or           (1)
         | xor          (2)
         | nand         (3)
```

If the codon being read produces the integer 6, then Eq. (3.1) gives $6 \ mod \ 4 = 2$, which would select rule `(2) xor`. In the derivation `<biop>` is replaced with `xor`. □

Each time a production from a rule with more than one production choice has to be selected to transform a non-terminal, another codon is read. In this way the system traverses the genome.

The mapping is deterministic, i.e. the same input sequence will map to the same output sequence if the grammar is unchanged, each time the same codon is expressed it will generate the same integer value. But depending on the derivation context, i.e. the current non-terminal to which the codon is being applied, a different production rule may be selected, this is called intrinsic polymorphism [106].

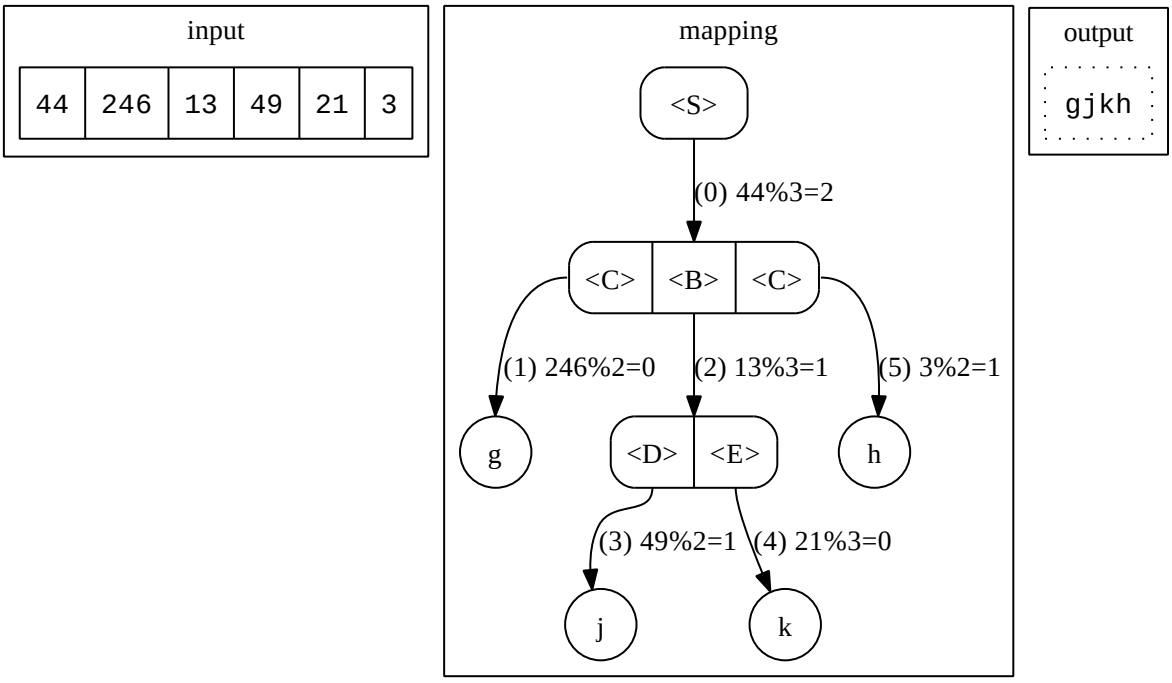


Fig. 3.3: Example of a derivation tree that generates a word, `gjkh`, using Grammar 3.2

While the mapping process in GE occurs and a sentence is being built, it can also be represented as a derivation tree. A concrete example of mapping in GE is shown in Fig. 3.3.

Example 7 (GE implementation in Python) Instead of pseudo-code, python code is presented, since it reads almost like pseudo-code. The implementation of a GE mapping is shown in Fig. 3.4.

3.1.4 Background to Mapping in GE

This section describes variations of mapping in GE. In GE the mapping, in combination with the operators, allows room for influencing the search bias. The representation, i.e. the encoding, as well as the operators can be changed in an attempt to make the search smoother [3]. However, Wagner and Altenberg [155] claim that complex gene interactions are advantageous for the chance of exploring new, functionally advantageous phenotypes,

```
def generate(input, max_wraps=1):
    used_input=0
    wraps=0
    output=[]

    unexpanded_symbols=[start_rule]
    while (wraps < max_wraps) and (len(unexpanded_symbols) > 0):
        # Wrap
        if (used_input%len(input) == 0) and (used_input > 0):
            wraps += 1
        # Expand a production
        current_symbol=unexpanded_symbols.pop(0)
        # Set output if it is a terminal
        if current_symbol[1] != NT:
            output.append(current_symbol[0])
        else:
            production_choices=rules[current_symbol[0]]
            # Select a production
            current_production=input[used_input%len(input)]%len(production_choices)
            # Use an input if there was more than 1 choice
            if len(production_choices) > 1:
                used_input += 1
            # Derivation order is left to right(depth-first)
            unexpanded_symbols=production_choices[current_production]+unexpanded_symbols

    #Not completely expanded phenotype
    if len(unexpanded_symbols) > 0:
        return (None, 0)
    else:
        return (output, used_input)
```

Fig. 3.4: Python implementation of GE mapping. Unexpanded non-terminal symbols are put on a stack and the input is used to determine which production will be chosen from each unexpanded non-terminal. The mapping is terminated when the stack is empty or the input is used up, if there are unexpanded non-terminals when the input is used the output is set to None (invalid). Python almost reads like pseudo-code.

```
<S> ::= <C>
| <C><C>
| <C><B><C>
<B> ::= <D>
| <D><E>
| <E>
<C> ::= g
| h
<D> ::= j
| k
<E> ::= k
| l
| m
```

Grammar 3.2: Example of a grammar for words.

i.e. evolvability as a mechanism of stabilization. Draghi et al. [35] claim that if the number of phenotypes accessible to an individual by mutation is smaller than the total number of phenotypes in the fitness landscape then mutational robustness can facilitate adaptation. This means that neutral diversity in a robust population has the ability to accelerate adaptation.

The bias in GE mapping that occurs when a production is selected from a rule with respect to the design of different grammars and grammar-defined introns has been studied by O'Neill et al. [116]. Wilson and Kaur [163] look at search, neutral evolution and mapping in evolutionary computation, especially at GE. The analysis is done by grouping the GE codons into quotient sets and showing their adjacencies regarding the mapping, this is then used to explain the population's movements on neutral landscapes. Here the equivalence relation of the quotient sets is the search neutrality of the codons, i.e. neutrality (many-to-one mappings) related to codons being indistinguishable on applying the mutation part of the evolutionary process. There are also results showing neutral evolution's effect on GE. Furthermore, it is shown that two phases of the mapping in GE, the bijective transcription of binary digits to integer input and the many-to-one translation of integer input to the mapped output belong to separate equivalence classes. The neutrality and the genome are

investigated and described as trivially neutral or not derivation neutral. Also cases are presented where rearranging the rules of a grammar does not affect performance.

The locality of a genotype-phenotype mapping describes how well genotypic neighbors correspond to phenotypic neighbors [138]. The locality of the mapping in GE has previously been investigated [138], the study concluded that some operators in GE had low locality, i.e. genotypic neighbors did not correspond to phenotypic neighbors. Montes de Oca [99] modifies the fitness function in order to improve results that are impeded by the the bias of a digit concatenating grammar. This illustrates the use of other search biases in addition to the grammar bias.

Grammar Alterations

There have been several studies regarding the expressiveness of the grammar and how it can be altered [see 19, 108, 25, 4, 26, 32, 117, 116, 119, 57]. McConaghy and Gielen [92] investigated how to use canonical form functions for genetic programming to evolve human interpretable functions, trying it on circuit modeling problems, and used grammar-defined introns in one of the experimental setups.

Grammars other than CFGs have been used or generated with GE:

Attribute grammars [25] Use an attribute grammar to solve knapsack problems. An attribute grammar defines attributes for the productions.

Christiansen grammars [123] Extend the attribute grammar to Christiansen grammars. The Christiansen grammars are adaptable, i.e. they can be modified while they are being used.

Logic Programming grammars [72] apply GE to adaptive logic programming, as an alternative search system for logic programming.

L-System grammars [122] created a system called genr8 which uses GE to generate L-systems for generating surfaces.

Set grammars [109] use GE with set grammars to evolve shelters by generating 3D-shapes.

TAG grammars [101] used tree adjoining grammars in conjunction with GE.

Grammar Mapping Variations

Different grammar mappings have also been tried

mod and bucket [74] An alternative mapping function for GE, this is called the Bucket Rule, differing from the standard modulo rule. The aim is to remove the fact that each codon value always codes for the same production choice if the number of production rules are the same, regardless of the rule. Instead, a mapping is done that allows the same codon value to code for different production choices depending on the rule, in order to remove the effect of rule definition ordering biasing the search. It is shown that by using the bucket rule the rule definition ordering bias is reduced.

Chorus [140] A position-independent encoding system for grammar-based EAs inspired by how genes produce proteins that regulate the metabolic pathways of the cell. The phenotype is the behavior of the cell's metabolism and this is mirrored in the development of the computer program in Chorus. In this procedure, the actual rule encoded by an gene (8-bit) is the same, regardless of the position of the gene within the genome. The values in the genome are moded by the total number of production choices in the grammar to guarantee that a vote is given every time a gene is read. The derivation is done left-to-right and to expand a non-terminal the production choices for that rule are considered and the production choice with the most votes is selected. Each time a production choice is encountered by a gene its vote is incremented by one, and each time it is used its vote is decremented by one. If the votes are tied the next gene is read. This makes the relative position of the genes important, not the absolute.

GAUGE [141] Genetic Algorithms Using Grammatical Evolution (GAUGE) is a position-

independent Genetic Algorithm that uses GE with an attribute grammar to dictate what position a gene codes for. A fixed set of codon pairs, one for each gene position in the original problem is used, with codons being 8-bit values. Moding the codons to get the appropriate value also gives a redundant coding, with values being the position the pair codes for and the value for that bit position. The mapping is done by first assigning generating position and valuing pairs with an attribute grammar and then putting the pairs in a correct order.

π-Grammatical Evolution [104, 37] A position-independent variation of input-output mapping, where the order of the derivation sequence is specified in the genotype. The papers showed significant improvement for the different derivation orders.

GE^2 In Grammatical Evolution by Grammatical Evolution(GE^2) a meta-grammar GE Algorithm, the input grammar is used to specify the construction of another syntactically correct grammar. The generated grammar is then used to generate a solution (see 5.2).

mGGA In the mGGA [111] the meta-grammar approach was shown as an alternative binary string genetic algorithm GA and improves its performance by the use of modules (see 5.2).

Reinforcement learning [97] Incorporates Q-trees, structures for maintaining a policy of actions that are appropriate for each state, to create Grammatical Evolution by reinforcement learning. The aim is to improve the individual's local search by incorporating Baldwinian learning, specific selection for general learning ability, i.e individuals who learn beneficial behavior fast are fitter. The effect is widened by introducing the Lamarck hypothesis, the idea that the parent's genome is changed during its existence. The learning is done for a number of episodes and then the Q-tree is reverse-mapped to the individual chromosome.

Representation of Genotype

It is not only the translation from genotype-to-phenotype that has been investigated. The representation of the genotype and the operators used will also add to the search bias. Hugosson [67] looked at genotype representations in GE by comparing binary and integer representations, finding support for integer representation. The paper deepens the investigation into the many-to-one mapping, and considers effects of the deterministic mapping from a linear sequence of input to output.

The previous research in GE has shown that there is a gap in the understanding of how grammar affects the search, and that the grammar is important for the performance. Even if operators and genotype representations are changed it is difficult to clearly distinguish these effects since they are affected by the mapping through the grammar.

3.2 Operators

Now we will present the different operations, the variations which are used to mix the solutions in the search as well as replacement and selection. This section will present the canonical operators in the GE search engine. The flow of the entire GE algorithm including selection, mutation, crossover, evaluation and replacement is shown in Fig. 3.5, GE implemented with a GA used as a search engine. From the original population a new population is selected. Crossover and mutation operators are applied to the selected population to create new individuals, which are evaluated. Finally, the original population is replaced in part or entirely by the selected population.

Alternative search engines to the canonical GA have also been applied to GE. First, the Particle Swarm algorithm was combined with GE to Grammatical Swarm [107]. Differential Evolution has also been used as a search engine to create Grammatical Differential Evolution [110].

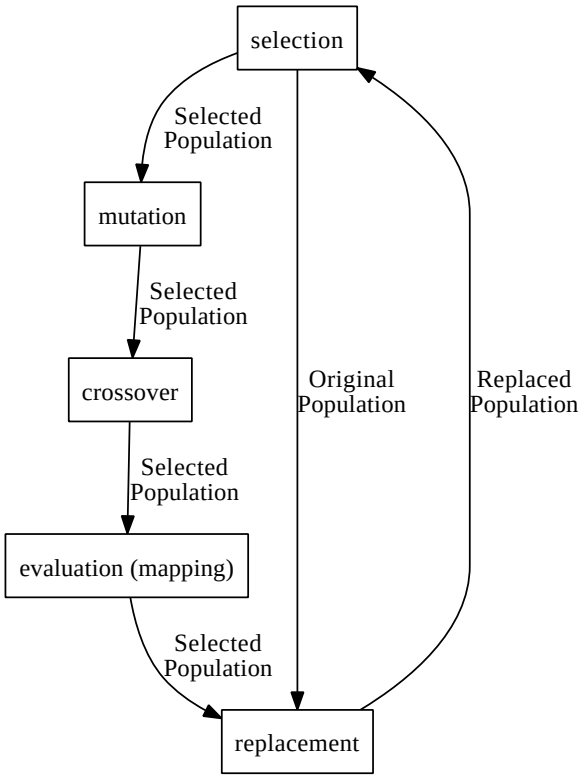


Fig. 3.5: GE implemented with a GA used as a search engine.

3.2.1 Variation Operations

The use of a mapping process creates a distinction between the search and the solution space. The genotype is evolved without knowledge of their phenotypic equivalent. When a change in the genotype occurs, this has been shown to create a ripple effect [71], as the function of the gene depends on the genes that precede it. Thus, a small genotypic change can lead to a large phenotypic change [138].

Crossover

Single point crossover in GE is performed as in GAs and is shown in Fig. 3.6. One point in each parent's genotype is selected. The parts on each side of the point are joined to the opposing part from the other parent. This crossover creates two children consisting of one part from each parent. Harper [50] looks at structure-preserving crossover operators and

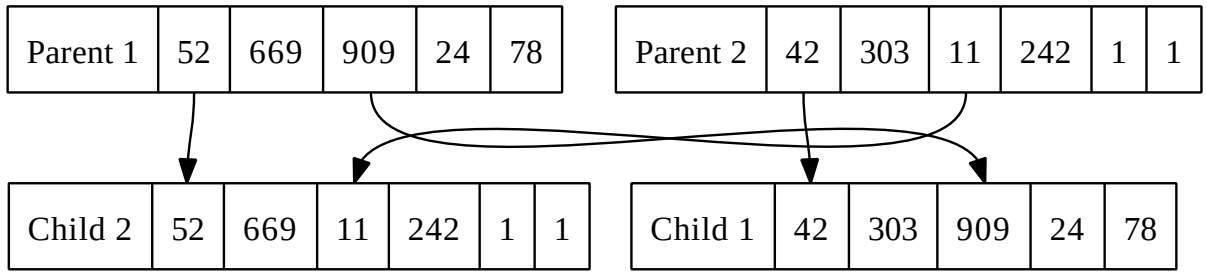


Fig. 3.6: Single point crossover in GE

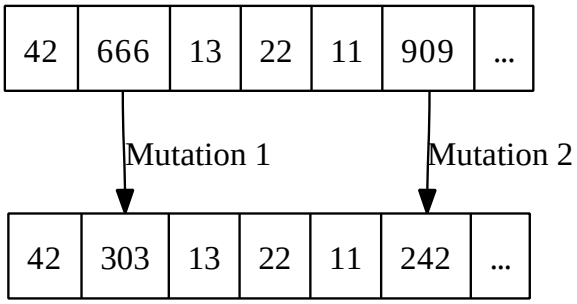


Fig. 3.7: Integer flip mutation in GE

self-selecting crossover operators [51]. A structure-preserving crossover operator preserves the derivation tree in order to reduce the disruption of the linear mapping.

Mutation

Integer flip mutation in GE is shown in Fig. 3.7. Each input codon has a uniform probability of changing to a new uniform integer value.

3.2.2 Selection and Replacement

Selection and replacement are often the standard GA types, tournament or roulette wheel selection, and steady state or generational replacement.

Selection

In the selection step, some individuals from the population are chosen according to some measure. Then the variation operations are applied to the selected individuals. In roulette wheel selection the probability for an individual to be selected is its fitness in relation to the others. Each individual gets a proportion of a roulette wheel equal to its fitness.

In tournament selection, a tournament size is chosen, and a number of individuals equal to the tournament size are randomly chosen from the population to compete in the tournament. The individual with the best fitness of the individuals selected for the tournament wins the tournament and is selected.

Replacement

The search progresses by replacing some individuals in the old populations with the newly created individuals. If generational replacement is used, the entire population is replaced, this search might converge quite slowly. A higher rate of convergence can be achieved if only the most fit new individual replaces the least fit old individual, if it is fitter.

This section has introduced the canonical GE operators, the variation operators mutation and crossover, and the selection and replacement operators.

3.3 GE in Practice

This section presents different applications of GE as well as implementations of the GE-algorithm.

3.3.1 Applications

Here follows a brief overview of different applications of GE from the wide range of areas where it has been applied. This shows the wide applicability of using a grammar in the algorithm. Gavrilis and Tsoulos [41] used GE for Fetal Heart rate monitoring as an application within medicine. On the more biological side Motsinger-Reif et al. [100] evolves

neural networks using GE to detect gene-gene interactions in the presence of error. White et al. [162] compares different GE strategies in human genetics.

For design, especially emergent design, O'Reilly and Hemberg [122] used GE for integrating generative growth and form exploration. Further exploration of GE in the arts was done when Abu Dalhoum et al. [1] generated music using GE. Reddin et al. [132] used GEVA to evolve elevator music. In another creative application Cebrian et al. [22] used GE for automatic plagiarism detection. Another well-studied area is finance [15, 34].

For more computer science related applications caching algorithms were investigated [112]. Also Adaptive logic programming was attempted using GE [72]. In corporation with other algorithms McKinney and Tian [95] uses GE to generate artificial immune systems. For feature extraction for time-series classification GE was used by Eads et al. [36], who created a two stage algorithm using grammars to constrain the set of valid feature extraction programs, and incorporating domain knowledge. In computer graphics Murphy et al. [102] evolved horse gaits, using GEVA.

3.3.2 Implementing GE

This section covers some of the published GE algorithms, for a more comprehensive list of GE implementations see `http://www.grammatical-evolution.org`. libGE, a C++ library for GE was reviewed by Wilson et al. [166]. Another tool is GDF [152] a tool for function estimation through grammatical evolution implemented in C++. JCLEC [153] is a Java framework for evolutionary computation that has a package for GE. Also Georgiou and Teahan [42] has implemented a complementary GE system in Java.

Grammatical Evolution in Java (GEVA) [118] is an open source implementation developed at UCD's Natural Computing Research & Applications group. In addition to providing the characteristic genotype-phenotype mapper of GE, a search algorithm engine and a simple GUI are provided. Furthermore, a number of sample problems and tutorials on how to use and adapt GEVA has been developed.

3.4 Summary

This chapter has described GE in order to lay the foundations for an examination of grammars in GE. We introduced the biological inspiration for GE, then examined grammar representation, the mapping and previous research in mapping. The most common operators used in GE were presented. Finally, application areas of GE and implementations of the GE algorithm were briefly introduced.

This chapter has presented gaps in the understanding of the mapping order as well as how grammars affect the search. Furthermore, the gaps concerning understanding how the grammar affects the performance of GE have been shown. Several studies have investigated grammars, which has highlighted the importance of grammars for performance in GE. Moreover, there have only been a few studies regarding the meta-grammars in GE and how they can adapt during the search. Most have studied properties and possibilities (expressions) of different grammars, and the bias that they allow. But much of the grammar mapping itself and the theory behind it are still unknown.

Now the preliminary part of the thesis has been completed, with grammars defined and the use of grammars in EC reviewed in Chapter 2, while this chapter presented GE. In Part II the exploration of grammars in Grammatical Evolution begins. Chapter 4 is concerned with the grammar mapping that occurs in GE and how this affects the search performance.

Part II

Experiments - Exploring Grammars in Grammatical Evolution

In Part II we start our empirical exploration of grammars in GE, studying performance and adaptation. From Part I we know that there are gaps concerning our understanding of grammar mapping order, the larger meta-grammars and how the grammar affects the change of codons.

The question is how the grammar could be used to improve performance and how it could be altered. First we need to understand how a grammar works in practice and this is pursued in Chapter 4. We studied the mapping order, which gives insight into the impact of the derivation order. It also shows how the grammar input is related to the rule order.

The meta-grammar studies investigate how a larger grammar, with a modified representation performs. Chapter 5 explores automatically defined functions for meta-grammars. The studies of the meta-grammars are extended in Chapter 6 to investigate not only the ability to capture modules, but also the scalability. Moreover, the impact of operators for the meta-grammar implementation as well as grammar design are studied in Chapter 7. These reveal that the meta-grammar scales well for regular problems of increasing size. In Chapter 8 the effects of the mGGA grammar design reveal that using building block structures has less variance than a grammar with a strong bias towards building block structures.

The theory is an extended discussion of the empirical results. The theory in Part III further investigates and tries to formalize and generalize the results that are discovered in Part II.

Chapter 4

Grammar Mapping

The first stop in the exploration of grammar in GE is the mapping process and how different grammars can alter it. This chapter explores the order of the GE mapping process and demonstrates how the grammar employed can be used to control the mapping order. Parts of this chapter have been published [58]. We studied the mapping order, which gives insight into the impact of the derivation order. It also shows how the grammar input is related to the rule order.

In Section 4.1 the different grammars are presented. Section 4.2 introduces the experiments, Section 4.3 shows the results, Section 4.4 contains the discussion and Section 4.5 summarizes the chapter.

4.1 Pre-, In-, Postfix Grammars

We investigate the importance of the ordering of the mapping process that occurs during the generation of a solution. Traditional GE constructs derivation trees depth-first, shown in Fig. 4.1. In πGE (see 3.1.4 on page 41), however, individuals can evolve the order in which non-terminals are expanded, leading to performance gains [14, 37]. This indicates that the order in which non-terminals are expanded can affect search efficiency. Other studies also indicate that grammar design can impact an algorithm's performance [61, 114, 103, 99]

```
<e> ::= ( <o> <e> <e> ) | <v>
<o> ::= +|-|*|/
<v> ::= x0 | x1 | <c>
<c> ::= 1|2|3|4|5|6|7|8|9
```

Grammar 4.1: Prefix grammars for Symbolic Regression, *italics* mark the difference between infix and postfix grammars.

```
<e> ::= ( <e> <o> <e> ) | <v>
<o> ::= +|-|*|/
<v> ::= x0 | x1 | <c>
<c> ::= 1|2|3|4|5|6|7|8|9
```

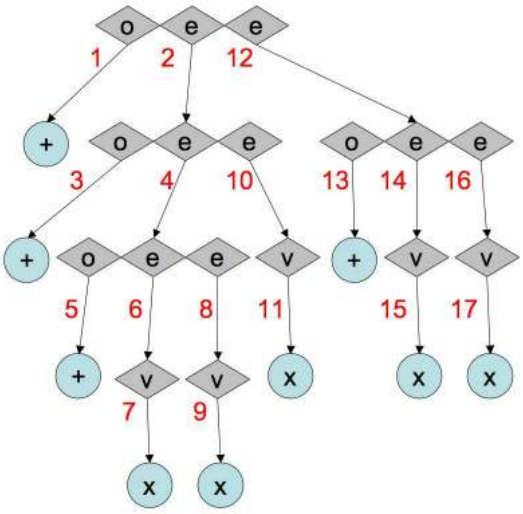
Grammar 4.2: Infix grammars for Symbolic Regression, *italics* mark the difference between prefix and postfix grammars.

Here we use the standard depth-first mapper, with three grammars which differ only in the ordering of the non-terminals in the productions. Grammar 4.2 is infix (typical in most previous GE work), Grammar 4.1 is prefix, and Grammar 4.3 is postfix. All these grammars are for symbolic regression problems. We then compare the performance of these grammars on a suite of symbolic regression problem instances. If the order in which non-terminals are mapped is truly important, we would expect differences in performance between the starkly contrasting prefix and postfix grammars.

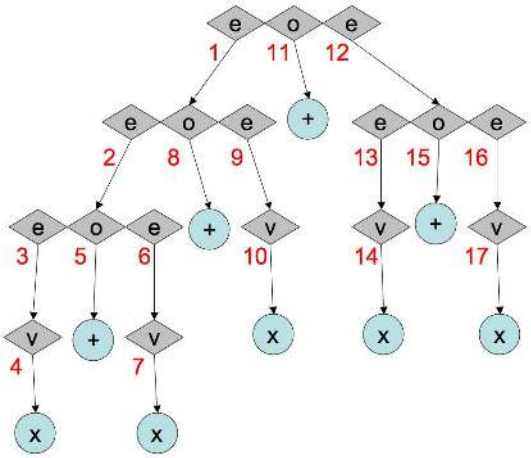
With prefix grammars for example, operators are determined earlier in the input sequence than the operands, whereas the opposite is true for postfix. As a result, the root of a syntax tree is the last component of a program that is determined in postfix, as opposed to the root being the first component of a program with prefix. See Fig. 4.1 where the grammars from Grammar 4.1 and 4.2 and 4.3 produce the derivation trees.

```
<e> ::= ( <e> <e> <o> ) | <v>
<o> ::= +|-|*|/
<v> ::= x0 | x1 | <c>
<c> ::= 1|2|3|4|5|6|7|8|9
```

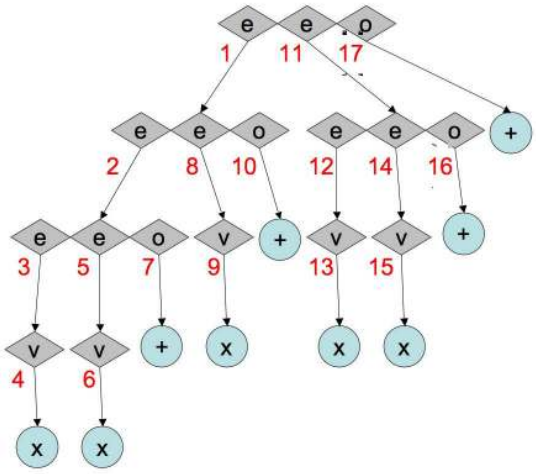
Grammar 4.3: Postfix grammar for Symbolic Regression, *italics* mark the difference between infix and prefix grammars.



(a) Prefix mapping



(b) Infix mapping



(c) Postfix mapping

Fig. 4.1: Derivation trees mapped from the different grammars from 4.1, 4.2 and Grammar 4.3. The grammars generate equivalent expressions (`x+x+x+x+x`) from different chromosomes of length 17 and the codon number is indicated in the figure. Diamonds denote non-terminal symbols and circles denote terminal symbols.

By using different grammars, the search space can be explored in different ways. The same genotype gives different derivation trees (in both content and structure) and phenotypes depending on the grammar. This is illustrated by examining the derivation trees that are created when mapping the genotype to the phenotype. Fig. 4.1 shows how different grammars can lead to different derivation trees that in fact represent the same phenotype

(the input sequences used to generate the trees, however, are different in each case).

4.1.1 Symbolic Regression

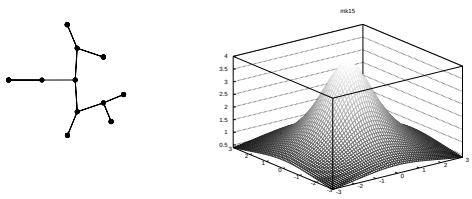
For a symbolic regression the goal is to find a function that matches a set of observed points from a target function. In this experiment the following target functions were used:

1. $8/(2 + x^2 + y^2)$
2. $x^3(x - 1) + y(y/2 - 1)$
3. $x^3/5 + y^3/2 - y - x$
4. $\frac{30*x^2}{(10-x)y^2} + x^4 - x^3 + \frac{y^2}{2} - y + \frac{8}{2+x^2+y^2} + x$
5. $\frac{30*x^2}{(10-x)y^2} + x^4 - \frac{4}{5}x^3 + \frac{y^2}{2} - 2y + \frac{8}{2+x^2+y^2} + \frac{y^3}{2} - x$

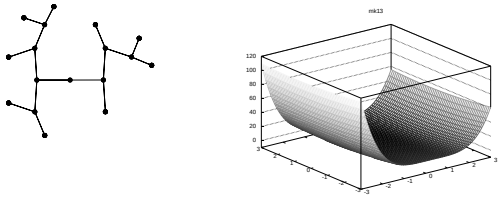
Some of these target functions were adopted from Keijzer [73], while others were created to encourage the evolution of larger expression trees. For each evaluation 20 random sample points for x and y were chosen from the range $[-3, 3]$. Fig. 4.2 shows the target functions plotted over this range, together with diagrams showing the structure of the target expressions, with the structural complexity increasing with each target. The source code for generating the trees is from Gustafson [48].

4.1.2 Grammar

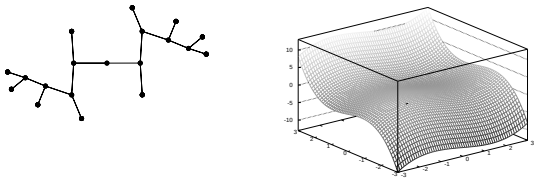
The grammars used are shown in Grammar 4.1 and 4.2 and 4.3. The only difference in the grammar is between the prefix-, infix- and postfix representation of the function expression. This means that the grammars have different sites that determine the order of the expansion of the grammar in relation to the root, see Fig. 4.1.



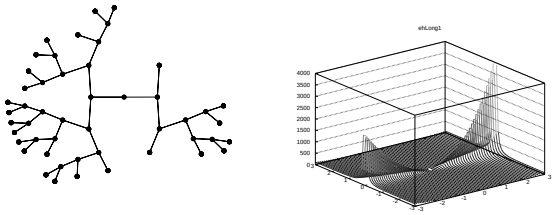
(a) Target (1), nodes=11, leaves=6



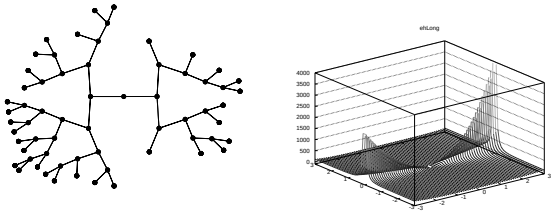
(b) Target (2), nodes=17, leaves=9



(c) Target (3), nodes=19, leaves=10



(d) Target (4), nodes=49, leaves=25



(e) Target (5), nodes=63, leaves=32

Fig. 4.2: Expression trees and a plot of the function over the range.

4.2 Experiment

The experiments are designed to test whether there is a difference in the performance between the different grammars. The performance is measured as the average best fitness ϕ after 50 generations over 1000 runs. The False Discovery Rate (FDR) [9] is calculated and the p-values are derived from two sided t-tests. The false discovery rate is used to correct for multiple comparisons. The FDR is the expected false positive ratevalue telling how many of the p-values from the multiple hypotheses that were significant given the significance level, α of the FDR-test.

Tab. 4.1: Parameters for the GE algorithm

Fitness function	See 4.1.1
Initialization	Ramped Half and Half
Grow Derivation tree depth	12
Selection operation	Tournament
Tournament size	3
Replacement	Generational
Elites	2
Population size	500
Max wraps	1
Generations	50
Crossover probability	0.9
Mutation probability	0.01

Hypothesis

H_0**:** No difference in best fitness between the grammars, i.e. $\phi_{Pre} = \phi_{In}$, $\phi_{In} = \phi_{Post}$ and $\phi_{Pre} = \phi_{Post}$

H_1**:** A difference in best fitness between the grammars, i.e. $\phi_{Pre} \neq \phi_{In}$, $\phi_{In} \neq \phi_{Post}$ or $\phi_{Pre} \neq \phi_{Post}$

α**:** The significance level of the test is 0.05.

4.2.1 Setup

Parameter settings for the GE algorithm are listed in Tab. 4.1. The input (called *chromosomes*) were variable-length vectors of integers (4 byte integers). Our fitness measure is the sum of the squared error over the 20 points chosen from the target function. One-point variable length crossover was used and an integer mutation operator where a new value was randomly chosen. For division a naive protection was implemented, 0.0 was returned, i.e. it was still deemed valid, if the divisor equaled 0. An individual is invalid if the phenotype contains non-terminals after mapping. Invalids are given the worst possible fitness.

Tab. 4.2: p-values for the grammars on the different targets, the average best fitness and standard deviation are shown next to the grammar. *Italics* indicate a significant p-value.

Target (1)		
Grammar	**Infix**(2.6969 ± 0.8490)	**Postfix**(2.6742 ± 0.7739)
Postfix	0.640	x
Prefix(2.714 ± 0.8189)	0.640	0.640
Target (2)		
Grammar	**Infix**(358.90 ± 121.9458)	**Postfix**(364.53 ± 120.4808)
Postfix	0.293	x
Prefix(372.22 ± 116.4965)	*0.039*	0.227
Target (3)		
Grammar	**Infix**(55.22 ± 14.48962)	**Postfix**(55.15 ± 15.43094)
Postfix	0.920	x
Prefix(56.22 ± 15.05861)	0.200	0.200
Target (4)		
Grammar	**Infix**(906.7 ± 325.1407)	**Postfix**(722.1 ± 228.3647)
Postfix	*2e-16*	x
Prefix(957.7 ± 325.5681)	*1.2e-4*	*2e-16*
Target (5)		
Grammar	**Infix**(938.2 ± 340.6127)	**Postfix**(759.3 ± 238.2787)
Postfix	*2e-16*	x
Prefix(955.9 ± 332.0197)	0.200	*2e-16*

4.3 Results

The best fitness over time is shown in Fig. 4.3 and in Fig. 4.4(a) box plots of the runs are shown. By examining the last generation of the runs for each problem with pairwise t-test between the different grammars their performance is compared, shown in Tab. 4.2.

For Target (1) there is no significant difference between any of the grammars. Infix is significantly better than Prefix for Target (2). There is no significant difference for Target (3). For Target (4) Postfix has significantly better performance than both Infix and Prefix, moreover Infix also has significantly better performance than Prefix. There is a signficant better performance for Postfix compared to Infix and Prefix for Target (5).

For the two larger problem instances, Targets (4 & 5), a performance advantage was observed for postfix when compared to both infix and prefix. Additionally, for Target (5) infix outperformed prefix.

When studying the results from Fig. 4.4(b) one can notice that postfix grammars always have more valid individuals when compared to prefix, except for Target (1), although for Target (2 & 3) the number of invalids in all grammars is close to zero, but for Targets (5 & 4) the difference is higher.

4.4 Discussion

All grammars show a similar behavior when it comes to fitness. An inspection of the results for each run revealed that for the prefix grammar, a significantly larger number of invalid individuals was generated after the initial population. Clearly, this can account for some of the differences in performance observed, but it is interesting to ask why so many invalids are being generated? More invalids mean less fitness evaluations performed. One explanation could be the different locations of the grammar expansions in the input string.

Example individuals at the last generation for Target (4 & 5).

```
Target(4)
Infix - Fitness:1014.33
((x1+(((((x1*(((x0*( (4.0+(x1+x0))+((x0*x1)-(x1+x1))))/x0)/5.0
))/4.0)/(5.0+(x1*(x1*x0))))/x1)/x1))*x0)

Postfix - Fitness:1015.75
(x1(((x1(((x1x0-)(x1x1 -)-)((x0x1/)(x0x1*)-)+)+)((x0x1/)(x1x1*
)+)+)(x1x0+)*)/)

Prefix - Fitness:1006.35
(*(/(+x0(*x1x1))(-x1 x0))(*(+x1x0)(-x0(*x1(*(/x0(*x1x1))x0))) ))

Target(5)
Infix - Fitness:1030.78
((((x0*x1)-x1)-(1.0-( x0/(7.0-((x1+(((x1+(x1*((x1/6.0)*(x1+(((
8.0+6.0)-x1)+(x1+(x0+2.0)))))))*((((7.0*
6.0)+x1)/x0)/(6.0*x1)))*2.0))/(x1/6.0)))) ))*((x0*x1)-x1))
```

```
Postfix - Fitness:1005.93
((((x0x0*)((x0((6.0(( x0(x1x1-)-)x0*)-)((((((x0((x1x1+)7.0/)*)
x0*)(x1x1/)+)x0/)x0*)(2.0x1/)*)-)*)9.0+) /)x0/)x0/)

Prefix - Fitness:1010.74
(+(*(+(-(/x16.0)x1)x0 )x0)(+5.0x0))
```

When studying the examples from the Fig. 4.1 is possible to see that the index of operators from <o> ::= +|-|*|/ are different. From Fig. 4.1 it is also possible to see that the index of the codon determining the last <exp> non-terminal in Fig. 4.1(a) and Fig. 4.1(b) is 16 while for Fig. 4.1(c) the index is 14. In the prefix solution the sum of the expansion index of terminals is lower than for infix, which in turn is lower than postfix solution. This comes from the fact that the grammar $\texttt{<e>}_i$ non-terminal at the max i is different for each grammar.

Pre-fix $\texttt{<e>}_i$ ::= ($\texttt{<o>}_{i+1}$ $\texttt{<e>}_{i+2}$ $\texttt{<e>}_{i+3}$)

In-fix $\texttt{<e>}_i$ ::= ($\texttt{<exp>}_{i+1}$ $\texttt{<o>}_{i+2}$ $\texttt{<e>}_{i+3}$)

Post-fix $\texttt{<e>}_i$ ::= ($\texttt{<exp>}_{i+1}$ $\texttt{<e>}_{i+2}$ $\texttt{<o>}_{i+3}$)

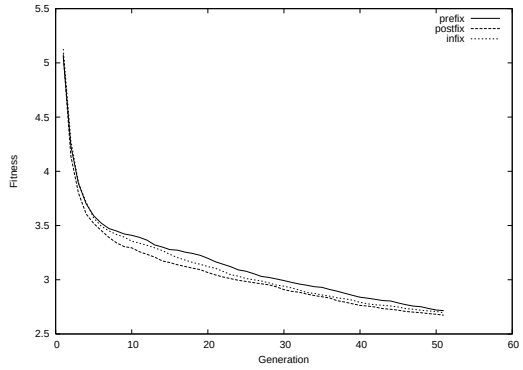
Since the max number of codons used for <o> is one this leads to an index difference for the codon which expands the last <exp> to be two between postfix and prefix, and between postfix and infix. The effect of changes to codons and their location will be further examined in Section 10.

4.5 Summary

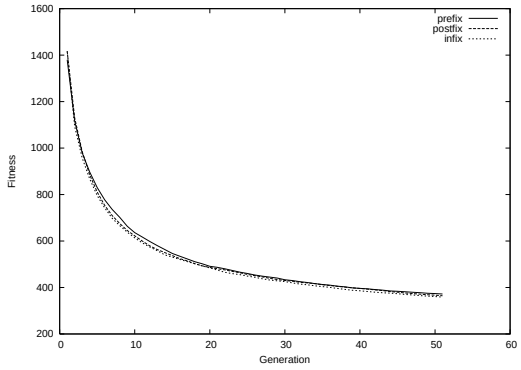
We wished to see if the order of symbols within a grammar can impact the performance of GE by comparing prefix-, infix- and postfix syntactical variants. The results suggest that the choice of grammar can produce performance advantage for two of the different

problems examined and no disadvantage for the others. This occurs because each grammar creates solutions of diverging shapes which react to the operators in different ways.

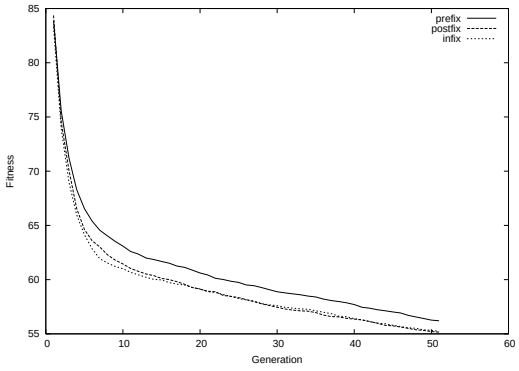
We have now examined a simple study of how the grammars can impact the search by altering the mapping order, and thus the neighboring phenotypes. We continue the investigation of grammars by studying how allowing the grammar itself to evolve might impact the search process. In Chapter 5 a meta-grammar approach to search is examined. By using a grammar that allows the structure of the grammar to change, this will allow the probabilities of the productions to change. First we study the principle of automatically capturing modularity, which is adopted from GP, and coupling this to an adaptive representation.



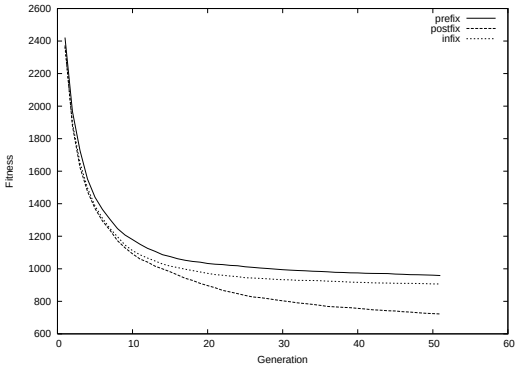
(a) Target (1), best fit.



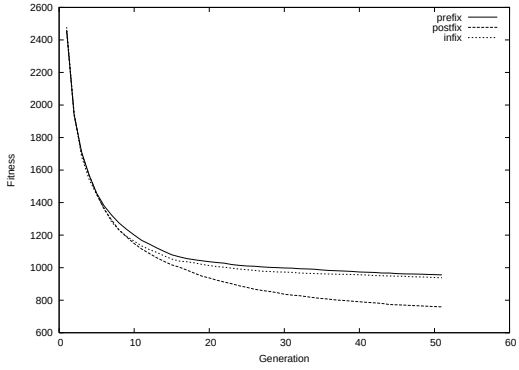
(b) Target (2), best fit.



(c) Target (3), best fit.

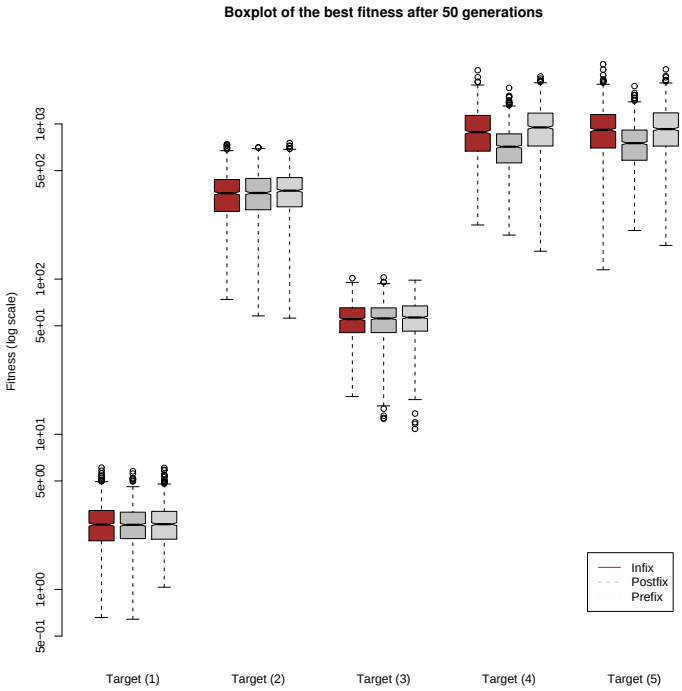


(d) Target (4), best fit.

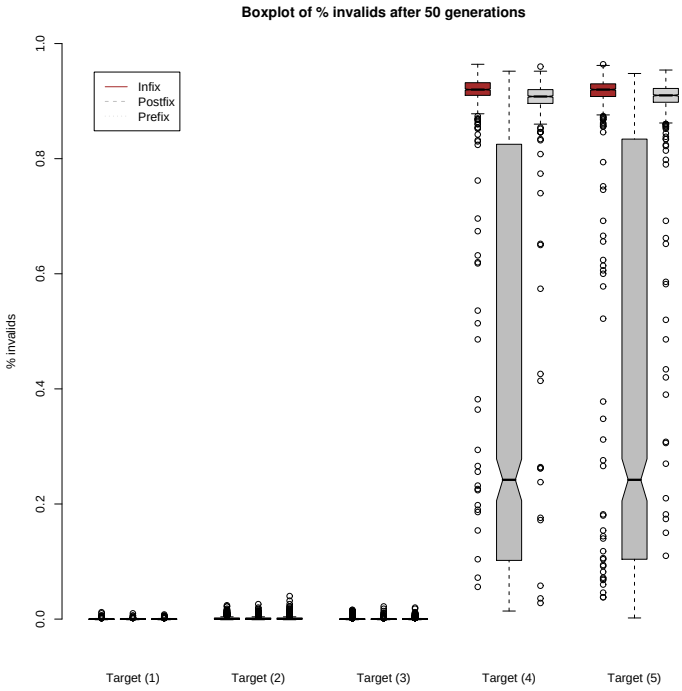


(e) Target (5), best fit.

Fig. 4.3: Best fitness results averaged over 1000 runs for pre-, in- and postfix experiments.



(a) Best fitness, y-axis is log scale



(b) % invalids in population

Fig. 4.4: Box plots of Best Fitness and % invalids at the final generation for pre-, in- and postfix experiments. For each target to the left is infix, center is postfix and right is prefix

Chapter 5

Meta-Grammar for Automatically Defining Functions - Modularity

After having studied the effects of grammar mapping order in GE we continue with studying how allowing the grammar itself to evolve might impact the search process. First we investigate how a grammar can be used to capture modularity. The grammar can be modified to allow definitions of structures in the rules and non-terminals, which will be able to bias the evolutionary search towards these structures. One approach to incorporate structures in a CFG is to use a meta-grammar. In a meta-grammar GE Algorithm the meta-grammar specifies the construction of another syntactically correct grammar. The generated grammar is then used to generate a solution.

This chapter investigates the principle of automatically capturing modularity, adopted from GP, and of coupling this to an adaptive representation. The contribution is the extension of this approach to grammatical GP systems by using dynamic definition of modules with fixed module signatures. Furthermore, this chapter also introduces a novel meta-grammar approach to modularity and compares this approach to other grammar-based approaches. The introduction of the meta-grammar and the ability to automatically capture modules change both the grammatical- and the evolutionary search bias. The contents are based on work presented by Hemberg et al. [62].

In Section 5.1 modularity is presented and in Section 5.2 meta-grammars in GE are introduced. Section 5.3 contains the experiments and the results. Finally, Section 5.4 recaptures this chapter.

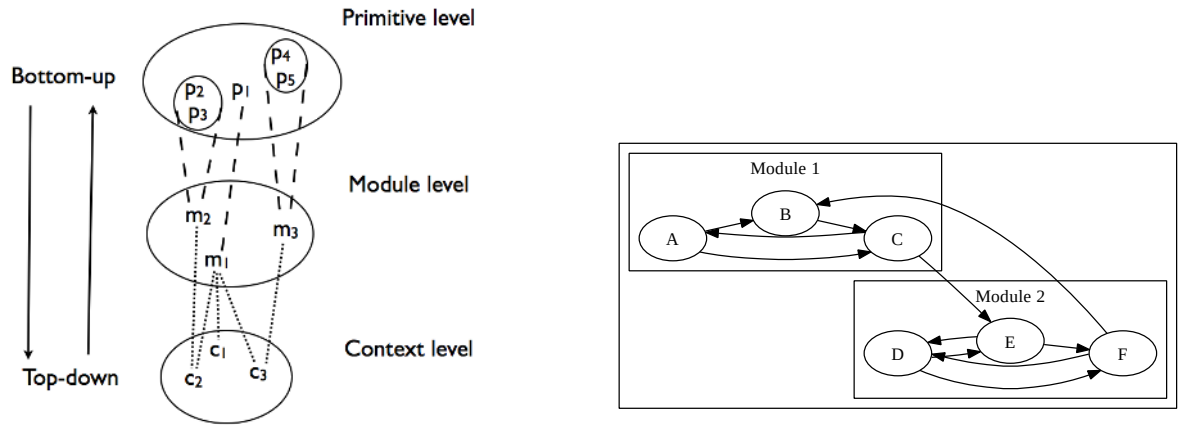
5.1 Modularity

This section provides an overview of research in modularity. Section 5.1.1 presents a brief overview of modularity, Section 5.1.2 introduces some examples of modularity definitions in EC, Section 5.1.3 further investigates some examples of modularity in GP, and finally, in Section 5.1.4 Automatically Defined Functions (ADFs) in GE are reviewed.

5.1.1 Modularity Overview

In EC literature, modularity is a reoccurring concept with a variety of definitions and approaches. The definition of modularity used in this thesis is quite general and often referred to, it is given by Simon [146] and it states that a module has more frequent interactions within the subsystem than outside the subsystem. An example of this is shown in Fig. 5.1(b). The initial consideration when discussing modularity is the context in which modules are defined. Modules and context are both abstract concepts and concrete objects, where the most general view is given by treating modules as abstract concepts that sometimes can be instantiated. One distinction of modularity can be drawn by a top-down or bottom-up view. The top-down view of modularity is the intuitive claim that abstract concepts and concrete objects can contain modules. Simon [146] generalizes the notion of modularity when talking about nearly decomposable problems. Fig. 5.1(a) shows a view of modularity broken up into primitive-, module- and context levels. It also shows how the bottom-up and top-down view relate to these levels.

The contents and relations in and between modules differ depending on the context. A module itself can be either an abstract concept or a concrete object (within a concrete object), or solely another abstraction in an abstract concept. An example of this can



(a) Modularity concept (b) Modularity example

Fig. 5.1: Fig. 5.1(a) shows a schematic view of the abstract concept of modularity. The context decomposes into modules, and modules contain primitives. Each level has its own measures and relations, also between the levels. The mappings between levels can have different properties. Fig. 5.1(b) shows a schematic view of the classification of modularity. The nodes (A,B,C) are classified as Module 1 and (D,E,F) as Module 2 since there are more edges between the nodes in the modules than between nodes in Module 1 and Module 2.

be seen in compression, where if something can be compressed it can be said to contain modules. Also worth noting is that the existence of modules implies the existence of different levels. Furthermore, there can be hierarchies of modules not only at the module level and primitive level. The components or primitives of the module exist at one level and the module at a different one. This does not discount elements from one level to exist, unchanged, at a different level. Moreover, each context can contain different modules. This top-down break-up does not give details about the primitive contents of the modules or how they are defined or related, only about their existence. It is possible to add measures to the modules, but some context-relevant measures and relations are difficult to find, and here the module serves a more descriptive and intuitive purpose.

The other view, non-exclusive of top-down, is bottom-up, identifying parts (primitives) that can be combined to form a module. Here, one complication is to identify elements that can be combined, as well as identifying the relations between the elements that create a module and how to distinguish sensible modules. In order to identify a module the relations between the elements must be measurable. Sometimes the measure of the modularity is

extended to the context level. One way to extend the bottom-up view of modules is to see them as relations between elements in a set. This is the basic requirement for modules. Variations of module definitions are then dependent on how these interfaces or relations are defined, e.g. as edges between nodes in a graph, as pathways between cells, or as wires connecting components on a circuit board. The relations are applied to elements in the set, e.g. to nodes in a graph, to cells, or to electronic components on a circuit board. These relations can be such that the elements in the set are partially ordered. The primitive elements in the sets can be mapped to a different level and the new representation considered for module definition. Once a module has been defined, it can be reused if there are regularities, i.e. when using hardware components for a computer a standard interface is necessary in order to allow the modules to be reused and connected. It is also possible to add relations between the defined modules, thus creating new modules and building a hierarchy of modules.

Yu [170] reports that from graph theory modules are the partition of vertexes γ of a graph Gr into disjoint subsets. A graph is a set of vertexes (nodes) γ and a set of weighted edges ϵ and is denoted $Gr = (\gamma, \epsilon)$. Many different constraints can be specified for the partitioning of nodes. The measure of a module in graph theory can be the expected value of the entities in the adjecency matrix.

Now that we know the general definition of modularity we will see how it has been used in EC.

5.1.2 EC Examples of Modularity

In EC, Garibay [39] notes, modularity is a more general concept than building blocks, in that a module does not have to be related to fitness. In EC there have been several definitions of modularity, this section lists them. Tab. 5.1 has an example of a breakdown of EC definitions of modularity.

With the existence of links in the problem, modules can be created, and in a fixed length context they should preferably be situated close to each other, in the GA literature

Tab. 5.1: Definitions of modules. The table intends to show the scope of module definitions. In order to form a module in a context there must be a set of elements and relations between the elements. Depending on the view of modules and complexity of elements the existence of relations is explored. TD is Top Down, BU is Bottom Up definition of modules

Author	**View**	**Context**	**Elements**	**Relations**	**Measure**
Simon [146]	TD/BU	"General"	Parts	Connections	Inter/Intra Connections
Yu [170]	BU	GA	Parts	Connections	Dependency Stricture Matrix(DSM)
Garibay [39]	TD	GA	Genomic & System	Dependencies	Fitness
Hornby [66]	TD	Evo Design Systems	Elements	Manipulation	Unit behavior
Koza [81]	TD	GP	Sub problems	Addable	Fitness
Woodward [168]	TD	GP	Functional and Terminal	Set definition	Previous definition
Watson [156]	BU	EC	Variables	Dependencies	Relative fitness
Toussaint [151]	BU	EC & Neural Networks	Functional traits' index set	Adaption covariance	Adaption covariance
De Jong et al. [30]	BU	EC co-evo representation	Variables	Dependencies	Relative fitness
Luerssen and Powers [90]	BU	Grammar GP, design	Variables	Variables' configurations	Fitness
Lipson et al. [88]	BU	EC Design	Design units	Performance	Independent performance
Wagner [154]	BU	Developmental Biology	Genotype	Pleitropy	Pleitropic effects
Chen et al. [23]	BU	GA	Sequence in $\mathbb{Z}_2$	Dependencies	Fitness
Parent et al. [124]	BU	GP	Variable	Links	Dependent collection
Shan et al. [144]	BU	GP(building blocks)	Subtrees	Fitness	Number of occurrences
Krawiec and Wieloch [83]	TD/BU	GP(functional modules)	Subtrees	Fitness	Monotonicity degree

these are called building blocks, see Goldberg et al. [47]. Since the introduction of GAs, the search for building blocks has been ever-present. Building blocks can be considered as compositions of genes with either more or less linkages between them according to Toussaint [151]. The goal is to find linkages between the GA variables that are fit and then are propagated through the search, see Goldberg [46]. Parent et al. [124] say dependent collections of links could be seen as modules and that modularization has mainly been of interest to the GP community but is related to the search of building blocks in a GA. The messy GA allows variable length strings that may be under- or over-specified in regard to the problem solved and positional flexibility, with the aim to group dependent variables into tight linkages [47].

Problems decomposed into components have different interactions, modularity is the interaction between components, interacting modules form levels and overlap of component use in modules [170]. In product design and development Dependency Structure Matrix (DSM) clustering is a matrix representation of a graph containing information of pair-wise interactions between every pair of components in a system. The aim of DSM-clustering is to transfer the pair-wise interaction information into higher-order interaction information. Yu [170] uses a method based on Minimum Description Length (MDL), using the minimal total length for model description and data mismatch, to detect clusters in the DSM.

In a paper by Wagner [154], from a biological view, modularity is defined as follows

> "Independent genetic representation of functionally distinct character complexes can be described as modularity of the genotype-phenotype mapping function. A modular representation of two character complexes C1 and C2 is given if pleiotropic effects of the genes are more frequent among the members of a character complex than among members of different complexes."

It states that modules are important in evolution as disassociated semi-autonomous units. Wagner [154] says that developmental biology considers disassociability and that development is semi-autonomous, i.e. how genes group into gene nets with different gene actions and products.

In addition, when Garibay [39] investigates the effects of modularity on the search space it is noted that modularity can bring improved scalability by using a compressed representation once the modules are defined. His definition of modules is as genomic primitives containing system primitives and other low level modules, this relates to building blocks in GAs, but differs as to how they are affected by genetic operators. Furthermore, by allowing modules the search space can also increase in size, and it should be taken into account that there are also "bad" modules, in contrast to the "good" ones [39].

In a slightly different context, with a view to engineering, when talking about evolutionary design, Lipson et al. [88] define modularity as "the separability of a design into units that perform independently." For Evolutionary Design Systems Hornby [66] defines modularity as "an encapsulated group of elements that can be manipulated as a unit". Relating it to the building block hypothesis in GAs, Hornby states that modularity is measured by the number of structural units in a design. To measure modularity the amount of procedure calls is counted. Additionally, when studying the development of co-evolutionary representation De Jong and Oates [28] define modularity as the property that several variables in a problem are dependent on one another as to their (near-)optimal settings, while the dependencies between the module and variables outside the module are weak compared to the former dependencies.

The modules are defined as regular and hierarchical, using an algorithm that in a bottom-up fashion creates modules, by comparing potential building blocks to different combinations. Watson [156] defines modularity when talking about compositional evolution as follows

> "In a given system of variables, the configuration of a subset of variables that maximizes the fitness of the system may depend on the setting of the remaining variables in the system. A system can be understood as modular if it can be described in terms of subsets of variables where the number of different configurations for a subset that could give maximal fitness (given all possible configurations of variables in the remainder of the system) is low."

and finds it important to consider the interdependency between modules.

Toussaint [151] examines the evolution of genetic representations and modular neural adaptation. The reducibility of the representation used to define elementary particles in physics is analogously used to describe the notion of functional modules in EC and neural adaptions. The modules are measured by the adaption co-variance.

After this overview of modularity and definition in EC in general we will now review modularity in GP in particular.

5.1.3 Modularity in GP

In GP, Koza [81] takes a top-down view of modularity and devotes an entire book to subdividing problems and solving the sub problems; if something is decomposable it consists of modules. A more bottom-up definition of a model, apart from examples from various areas, would be difficult to find.

Angeline and Pollack [5] investigate the evolutionary induction of subroutines and introduce a Genetic Library Builder which allows compression of randomly selected subtrees. The compressed subtrees are assigned unique names and placed in a library, which makes it available for other solutions. The subroutines in the library are evaluated by the extent to which they are used in future generations. Compressed subroutines can also be expanded and replaced explicitly in the individual.

An approach to more problem-specific code is introduced by Spector [147] The capacity of GP is increased by introducing Automatically Defined Macros (ADM) that perform source code transformations, which allows implementation of new control structures.

From the GP context, other approaches to modularity are O'Reilly [121], who looks at the generality of Automatically Defined Functions (ADFs), an approach to capture modules and regularities using GP, by applying ADFs to Simulated Annealing. Whigham [160] biases the search in GGP by modifying the grammar trying to identify a production that appears to be useful and encapsulating it as an expansion in the new grammar. This creates a global change to the grammar.

The Push programming language is designed for the expression of evolving programs with EC and automatically provides multiple data types such as automatically defined subroutines, control structures and architecture. Spector and Robinson [148] use the Push in combination with GP to perform auto constructive evolution. Yu [169] investigates hierarchical processing for evolving recursive and modular programs using higher-order functions and lambda abstraction, concluding that appropriate higher-order functions are needed for beneficial structure abstraction.

Shan et al. [144] define building blocks as "sub-trees which appear more frequently in good individuals". Woodward [168] defines a module in GP as follows "A module is a function that is defined in terms of a primitive set or previously defined modules." When evolving encapsulated programs as shared grammars Luerssen and Powers [90] talk about modularity as "A subset M of variables in a specific design problem can be called a module if the number of possible configurations of M that maximize the fitness for at least one configuration of the remaining variables is less than the number of all possible configurations for M."

A low level modularization technique for linear GP system based on compression is presented by Parent et al. [124]. The algorithm they use operates on the compressed individual where module identification is facilitated by regularity in the representation. Krawiec and Wieloch [83] study what they call functional modularity for GP. They use the intuitive GP module definition "a piece of program code (subtree)." and continue to analyze the modules using fitness cases and also introduce subgoals for the fitness evaluation. Monotonicity is used to assess the subgoals utility for searching for good modules, where for a given subgoal and a sample of modules, monotonicity measures the correlation of the subgoals distance to the modules' semantics as well as the fitness of the solution the module is part of. Monotonicity differentiates two problems with different modularity, allowing distinction of subgoals, and may be potentially used for problem decomposition. Kashtan et al. [70] found that varying environments can speed up evolution, especially when there are modularly varying goals. This suggests that varying environments might contribute to the speed of natural evolution.

5.1.4 Automatically Defined Functions in GE

In many examples of problem solving, we humans use a divide-and-conquer approach by constructing sub-solutions which may be reused and combined in a hierarchical fashion to solve the problem as a whole. GP provides the ability to automatically create, modify and delete modules, which can be used in a hierarchical fashion.

Some previous work with GE and modularity [113] has also been undertaken, where functions were defined by the grammar, similarly to Automatically Defined Functions [81]. Functions were dynamically created using a dynamic grammar approach that allowed specification of multiple functions and a variable number of arguments for each function [52]. The newly created ADFs were dynamically appended onto the core grammar in such a manner that it was possible to invoke them from the main function.

This section has reviewed the concept of modularity in EC and in particular modularity in GE, as well as the studies of Automatically Defined Functions in GE. This has shown that there are several ways to define and promote modularity. In the next section we introduce a meta-grammar approach to define modules in the grammar and bias the search towards these modules.

5.2 Meta-Grammars and Grammatical Evolution

This section presents meta-grammars in GE and one implementation, Grammatical Evolution by Grammatical Evolution.

Adaptation, alterations of a solution from experience, which can be used by succeeding solutions, can be seen as learning. Learning and adaptation are useful for problem solving since they allow the search to progress towards a more optimal solution. The grammar biases the search to different regions of the search space.

Problems can be modular. With modularity a solution might find an underlying structure of a problem. Moreover, the speed of the search can be improved if there are modules. Altering all the parts of a module can be avoided, and only the connections between mod-

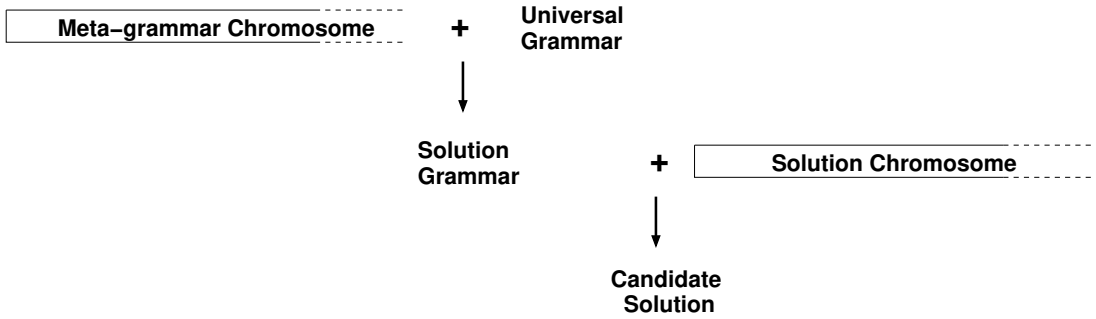


Fig. 5.2: An overview of the meta-grammar approach to GE. The meta-grammar generates a solution grammar, which is used to generate a candidate solution.

ules can be altered. The grammar can consist of modules as well as bias the search towards modular solutions.

Having the ability to learn by modifying the grammar itself means that the grammar can be used as a form of memory, for example to aid learning in a changing environment.

5.2.1 Grammatical Evolution by Grammatical Evolution

This section describes the Grammatical Evolution by Grammatical Evolution (GE^2) algorithm [114], which is in turn based on the GE algorithm [115]. In a meta-grammar GE Algorithm the input grammar is used to specify the construction of another syntactically correct grammar. The generated grammar is then used to generate a solution. This process is illustrated in Fig. 5.2.

The proposed representation aims to improve identification of modules in a problem. A variable length genotype is used to create a fixed length phenotype. In order to allow evolution of a grammar, another grammar must be provided to specify the form a generated grammar can take. By allowing an Evolutionary Algorithm to adapt its representation (in this case through the evolution of the grammar, via the evolution of the genotype) it can provide the population with enhanced robustness in an environment that changes over time, as well as with an ability to automatically incorporate bias into the search process.

```
<g> ::= <function_definitions>
        "<code> ::= <line>
                  | <code><line>"
        "<line> ::= operation
                  | function_call"
<function_definitions> ::= <function_code>
                         | <function_code><function_definitions>
<function_code> ::= <function_line>
                  | <function_line><function_code>
<function_line> ::= function_operation
```

Grammar 5.1: Simple meta-grammar example for evolving multiple functions. Note that `<code>` and `<line>` are quoted.

Meta-Grammar Mapping

The GE^2 approach has two distinct grammars, the *meta-grammar* and the *solution grammar*. The notion of a meta-grammar is adopted from a universal grammar in linguistics and refers to a universal set of syntactic rules that hold for spoken languages [24]. The meta-grammar dictates the construction of the solution grammar. In this study, the genotype consists of two separate, variable-length, binary chromosomes, the first chromosome is used to generate the solution grammar from the meta-grammar and the second chromosome generates the solution itself. In Ex. 8 an example of the mapping in GE^2 is presented.

Example 8 (GE^2 mapping) In Grammar 5.1 a simple meta-grammar for evolving a different number of functions is shown. This grammar allows the meta grammar to generate one or more functions containing one or more `function_operation` terminals. These functions can then be called from the solution grammar by the `function_call`. Terminals in the meta-grammar that are "quoted" are non-terminals, or part of BNF syntax in the solution grammar.

To generate a solution, using a meta-grammar and GE^2, the genotype consists of two chromosomes, the meta-chromosome $C_m = \langle 0, 0 \rangle$ and the solution chromosome $C_s = \langle 1, 0, 0, 0, 1 \rangle$ and the meta-grammar G_m in Grammar 5.1. The meta-chromosome and the

```
0. <g>
1. <function_definitions> "<code> ::= <line> | <code><line>" "<line> ::= operation | function_call"
2. <function_code> "<code> ::= <line> | <code><line>" "<line> ::= operation | function_call"
3. <function_line> "<code> ::= <line> | <code><line>" "<line> ::= operation | function_call"
4. function_operation "<code> ::= <line> | <code><line>" "<line> ::= operation | function_call"
```

The derivation of the solution grammar given the solution chromosome is

```
0. function_operation <code>
1. function_operation <code><line>
2. function_operation <line><line>
3. function_operation operation <line>
4. function_operation operation function_call
```

Fig. 5.3: Derivation of meta-grammar and solution grammar from Grammar 5.1

meta-grammar give the following derivation of the solution grammar G_s in Fig. 5.3 Finally the phenotype(solution) is `function_operation operation function_call`. □

Operators in GE^2

Crossover in GE^2 operates between homologous chromosomes, with the meta-grammar chromosome from the first parent recombining with the meta-grammar chromosome from the second parent, the same occurs for the solution chromosomes. In order for evolution to be successful it must co-evolve both the meta-grammar and the structure of solutions based on the evolved meta-grammar, and as such the search space is larger than in standard GE.

Background

There have been a number of studies of a meta-grammar approach to GE [114, 111, 33, 31]. The original meta-grammar study [114] investigated the feasibility of this approach and its application in dynamic environments. In each of these the rate of evolutionary search was equal for both the meta-grammar and solution chromosomes by using the same rates of mutation and crossover. An observation of some solutions and solution grammars evolved by meta-grammar GE has shown a tendency to generate grammars that did cannot to produce many different strings [31]. **?** [**?**] looked at the size of the search space when simultaneously evolving grammars and finds the meta-grammar search space quite large. There have not been any studies combining ADFs in GE with GE^2.

```
<g> ::=
       <def_fun_u>
       "<prog>      ::= public Test() { while(get_Energy_Left()) { <code>} } "
       "<code>      ::= <line> | <code> <line>"
       "<line>      ::= <condition> | <op>"
       "<condition> ::= if (food_ahead()==1) { <line> } else { <line>}"
       "<op>        ::= left(); | right(); | move(); | adf*();"
<def_fun_u>    ::= <def_fun_s> | <def_fun_u> <def_fun_s>
<def_fun_s>    ::= "public void adf*() {" <adfcode> "}"
<adfcode>      ::= <adfline> | <adfcode> <adfline>
<adfline>      ::= <adfcondition> | <adfop>
<adfcondition> ::= if (food_ahead()==1) { <adfline> } else { <adfline> }
<adfop>        ::= left(); | right(); | move();
```

Grammar 5.2: Example Ant trail meta-grammar, adf_{mg}, is a meta-grammar that can evolve ant-trail solution grammars.

5.3 Experiments & Results

In this study we wish to determine if one of the three ADF representations for GE has a performance advantage across a range of benchmark problems.

5.3.1 Meta-Grammar ADF

Grammar 5.2 is an example meta-grammar used for the Ant trails. `adf*()` is a function call to a defined function, where a codon is used to select which function is called. In the above example quotes are used to escape symbols, e.g. to avoid expanding non-terminals in the meta-grammar and instead expanding them in the solution grammar. In a solution grammar which contains multiple ADF definitions the grammar is post-processed to make each function signature unique.

5.3.2 Setup

The representations are a GE grammar with the ability to define one method (adf), a GE grammar that can define any number of methods ($adf - dyn$) and a novel meta-grammar (adf_{mg}) approach. The control for the experiment is a standard GE grammar (std). None

Tab. 5.2: Parameter settings for the GE algorithm

Parameter	Values
Fixed chromosome size	100, (200 for normal GE)
Initialization	Random
Selection operation	Tournament
Tournament size	3
Replacement	Generational
Max wraps	1
Generations	50
Population Size	500
Elite Size	2
Crossover probability meta	0.9
Crossover probability solution	0.9
Mutation probability meta	0.05
Mutation probability solution	0.05, (0.05 for normal GE)

of the grammars allow ADFs to call ADFs. Unless noted 30 runs were made and the significance of the results is tested by a t-test with p-value=0.05. The settings are shown in Tab. 5.2.

The chromosomes were variable-length vectors of integers (4 byte integers) and had the same initial length. We used one-point crossover, where the same crossover point is used for both parents and integer boundaries are respected. The mutation was integer mutation, where a new codon value was randomly chosen. The meta-grammar generates the content of the ADFs and the number of ADFs that the solution grammar can use. Wrapping is used on both chromosomes; if the mapping is still incomplete the individual is invalid and is assigned the worst possible fitness.

Three different ant trails and a symbolic regression problem were tested. We will now describe the problem and the grammars for each case.

5.3.3 Ant Trails

The goal of the ant trails is to find a program for controlling the movement of an artificial ant in order to find all of the food lying on irregular trails on a two-dimensional toroidal

```
<prog>      ::= <code>
<code>      ::= <line> | <code> <line>
<line>      ::= <condition>  | <op>
<condition> ::= if(food_ahead()==1) {<code>} else {<code>}
<op>        ::= left(); | right(); | move();
```

Grammar 5.3: *std* - The standard GE grammar for the ant trails.

```
<prog>         ::= "public Ant() { while(get_Energy() > 0) {"<code>"}} "
                   "public void adf0() {"<adfcode>"}"
<code>         ::= <line> | <code> <line>
<line>         ::= <condition>  | <op>
<condition>    ::= if (food_ahead()==1) {<line>} else {<line>}
<op>           ::= left(); | right(); | move(); | adf0();
<adfcode>      ::= <adfline> | <adfcode> <adfline>
<adfline>      ::= <adfcondition>  | <adfop>
<adfcondition> ::= if (food_ahead()==1) {<adfline>} else {<adfline>}
<adfop>        ::= left(); | right(); | move();
```

Grammar 5.4: *adf* - GE grammar for the ant trails, with only one ADF.

grid. The ant can sense if there is food in the single square it is currently facing. The potential actions of the ant are turning right, turning left, and moving forward one square, all requiring one energy unit. There is a maximum amount of energy that the ant can use; after the energy is used the number of food left on the trail is counted.

Three Ant trails, the Santa Fe Ant trail, Los Altos Trail from [82] and San Mateo Trail [81] are tested. The Santa Fe trail is a 32×32 toroidal grid containing 89 pieces of food and 600 time steps. The Los Altos trail is a 100×100 toroidal grid, 157 pieces of food and 2000 time steps.

The San Mateo trail consists of 9 parts, each made up of a 13×13 grid containing different discontinuities in the food trails. The borders on each trail part are electrified, if the ant goes over the edge the current fitness case is terminated. The total number of food is 96 and it has 120 right or left turns and 80 moves on each part.

None of the ADF functions for the ant trails take any arguments. See Grammar 5.2 for the meta-grammar used. Grammar 5.3 and 5.4 and 5.5 show the grammars used.

```
<prog>         ::= "public Ant() { while(get_Energy() > 0) {"<code>"} }"<adfs>
<adfs>         ::= <adf_def>  | <adf_def> <adfs>
<adf_def>      ::= " public void adf*() {"<adfcode>"}"
<code>         ::= <line> | <code> <line>
<line>         ::= <condition>  | <op>
<condition>    ::= "if(food_ahead()==1) {"<line>"} else {"<line>"}"
<op>           ::= left(); | right(); | move(); | adf*();
<adfcode>      ::= <adfline> | <adfcode> <adfline>
<adfline>      ::= <adfcondition>  | <adfop>
<adfcondition> ::= "if (food_ahead()==1) {"<adfline>"} else {"<adfline>"}"
<adfop>        ::= left(); | right(); | move();
```

Grammar 5.5: adf_{dyn} - The grammar for the ant trails, which allows multiple function definition is shown below. `adf*()` is expanded to create unique signatures for the allowed functions. Then the function `adf*()` function call is used to determine which of the functions to call.

5.3.4 Results - Ant Trails

This section deals with the results from the different ant trails. For all the trails it is beneficial to use ADFs. $\overline{\text{overline}}$ indicates average value between the runs. The results are shown in Tab. 5.3

Santa Fe Ant Trail

A plot of the Santa Fe Ant trail is shown in Fig. 5.4. The average best fitness over the runs of the last generation for the different grammars is $\overline{std} = 37.90$, $\overline{adf} = 20.33$, $\overline{adf_{dyn}} = 18.63$ and $\overline{adf_{mg}} = 24.57$. In order to test the significance of the results a t-test on the last generation confirms that ADFs are significantly better.

Los Altos Ant trail

For the Los Altos Ant trail a plot is shown in Fig. 5.5. The average best fitness of the last generation for the Los Altos trail for the different grammars is $\overline{std} = 33.46$, $\overline{adf} = 12.38$, $\overline{adf_{dyn}} = 16.47$ and $\overline{adf_{mg}} = 17.57$ Performing a t-test reveals that runs with ADFs are performing significantly better in the last generation compared to the ones with no ADFs.

Tab. 5.3: p-values for the grammars compared to the standard grammar on the different trails, the average best fitness and standard deviation is shown next to the grammar. Underlined indicate a significant p-value.

San Mateo

Grammar	**Std**(37.900 ± 12.220)
$adf(20.333 \pm 8.222)$	$1.78e-08$
$adf_{dyn}(18.633 \pm 7.054)$	$4.59e-10$
$adf_{mg}(24.567 \pm 7.619)$	$4.34e-06$

Santa Fe

Grammar	**Std**(33.461 ± 12.747)
$adf(12.381 \pm 13.590)$	$0.00e+00$
$adf_{dyn}(16.470 \pm 14.934)$	$0.00e+00$
$adf_{mg}(17.567 \pm 15.051)$	$2.81e-09$

Los Altos

Grammar	**Std**(90.900 ± 7.411)
$adf(82.233 \pm 2.837)$	$1.46e-07$
$adf_{dyn}(82.967 \pm 2.953)$	$1.09e-06$
$adf_{mg}(83.667 \pm 3.604)$	$1.12e-05$

San Mateo Ant Trail

For the San Mateo trail, Fig. 5.6 the average best fitness of the last generation is $\overline{std} = 90.90$, $\overline{adf} = 82.23$, $\overline{adf_{dyn}} = 82.67$ and $\overline{adf_{mg}} = 83.67$. Also for this trail with slightly different behavior it is significantly better in the last generation to use ADFs. For all the Ant trails it beneficial to use ADFs.

5.3.5 Symbolic Regression

A number of fitness functions for symbolic regression were examined, they were inspired by [81]. The statically defined grammar allows grammars which take one argument, while

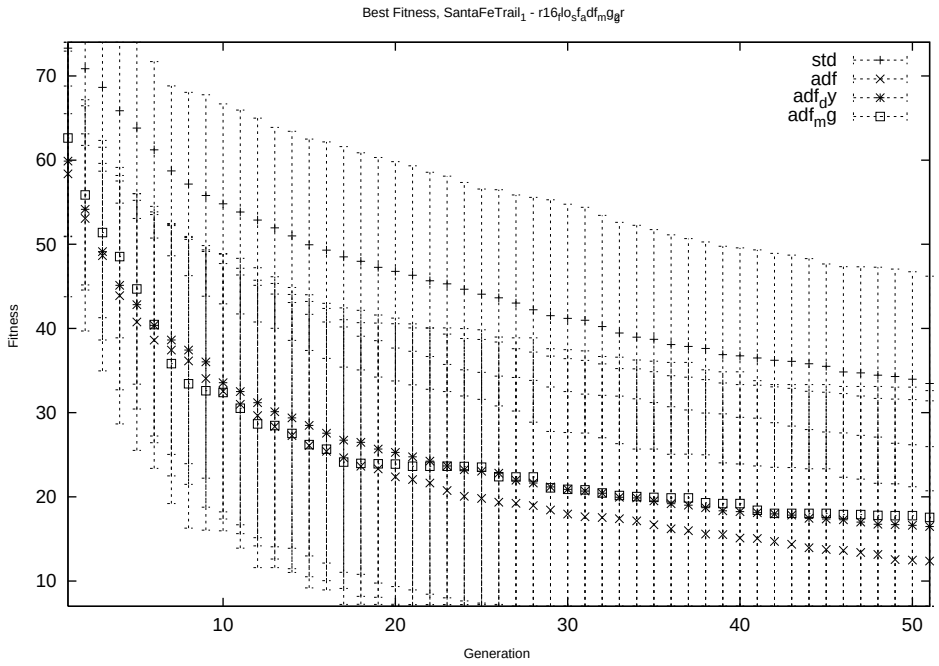


Fig. 5.4: Santa Fe Ant trail, average best fitness over the runs with error bars for each generation

the meta-grammar allows the defined methods to take a variable number of arguments.

$$x + x^2 + x^3 + x^4 + x^5 \tag{5.1}$$

$$x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^{10} \tag{5.2}$$

$$x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 \tag{5.3}$$

These are different degrees of a polynomial similar to the increase of function terms by Koza [81], shown in Fig. 5.7. The 21 fitness cases are selected from $[-1, -0.9, \ldots, 1]$. Fitness is the sum of the squared error for each fitness case. The `(GeneralRandomConstant)` generates 1000 samples in the range -1.000 to 1.000. All symbolic regression grammars use a protected division operator (d), 0.0 was returned if the divisor equaled 0.

To create dynamic problems two Symbolic Regression problems that change periodically are created. In the first the period is every 10 periods Eq. (5.1) switches between Eq. (5.2). In the second problem every 10 generations a polynomial one degree higher than the

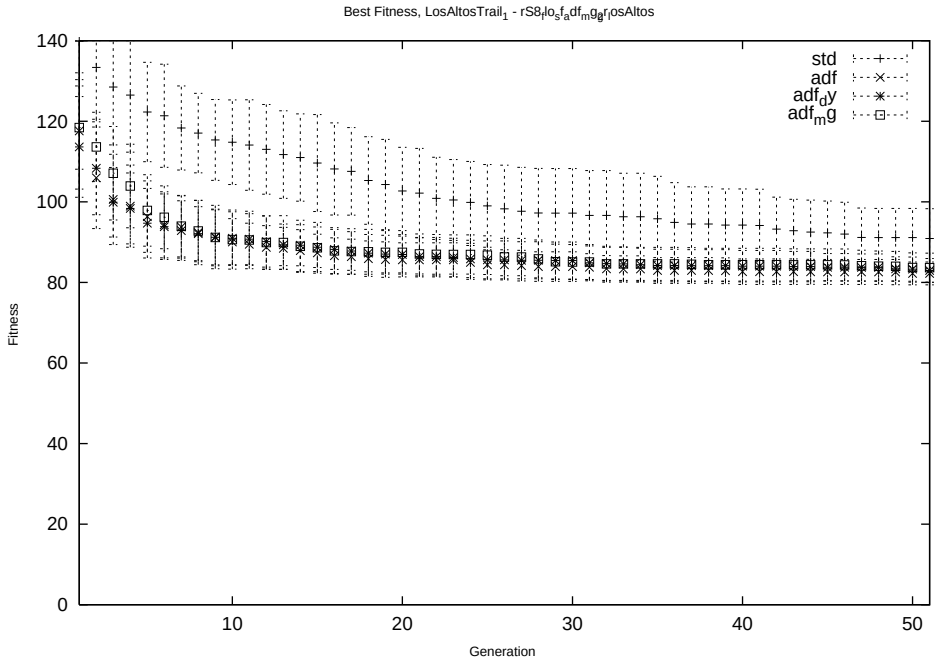


Fig. 5.5: Los Altos Ant trail, average best fitness with error bars over generations

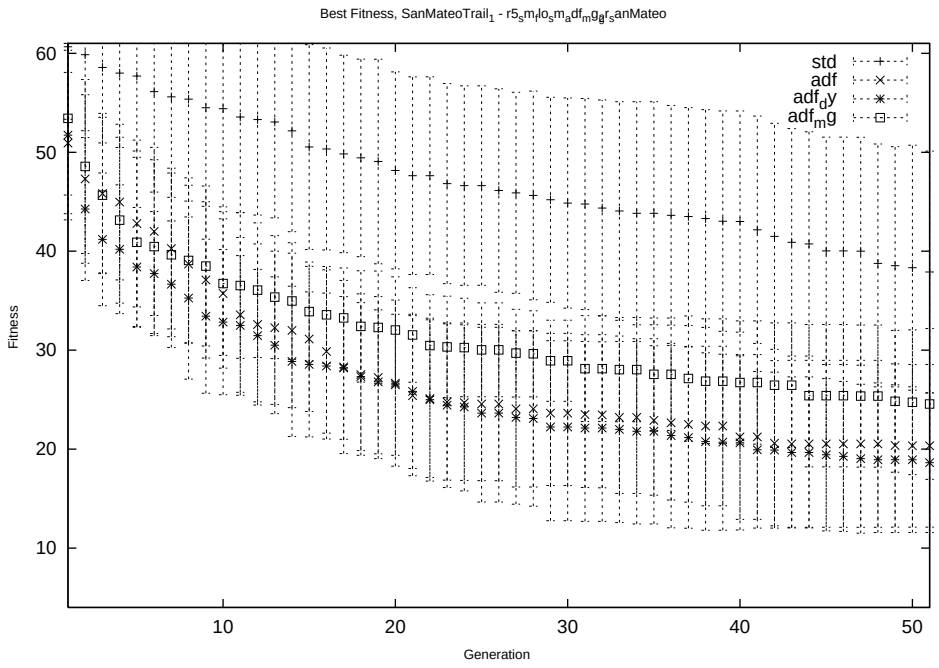
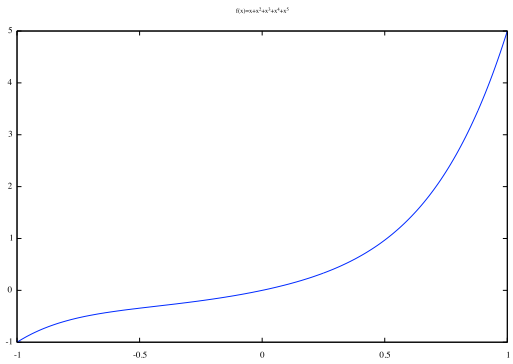
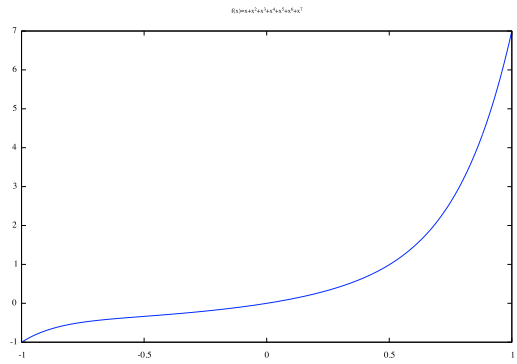


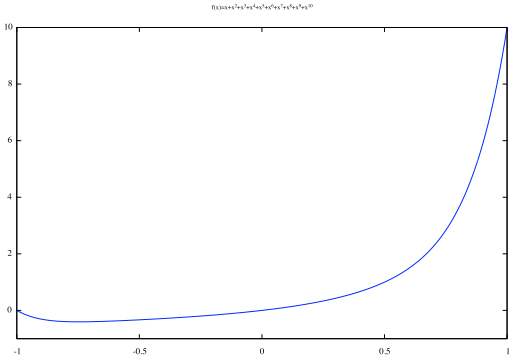
Fig. 5.6: Average best fitness plot with error bars over the generations for the San Mateo ant trail. A t-test confirms that the fitness differs significantly between standard GE and the other grammars for all problems in the final generation.



(a) $x + x^2 + x^3 + x^4 + x^5$, Eq. (5.1)



(b) $x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7$, Eq. (5.3)



(c) $x+x^2+x^3+x^4+x^5+x^6+x^7+x^8+x^9+x^{10}$, Eq. (5.2)

Fig. 5.7: Symbolic regression polynoms, $x \in [-1, 1]$

currently highest is added in Eq. (5.3) $f_0(x) = x$, $f_t(x) = f_{t-1}(x) + x^{degree(f_{t-1}(x))+1}$, $0 \leq t \leq$ generation/period. An added level of complexity occurs when not only the solutions are changed, but the optimal solution of the problem itself changes as well, a so called dynamic environment. This makes the task of finding a global optimum more temporal, and might best be described as "survival".

The meta-grammar approach for the symbolic regression problems allows creation of any number of functions with variable numbers of function arguments. The `ADFPARAM` and `ADFUSE` in Grammar 5.9 indicate where the grammar inserts and uses function arguments. `adf*(ADFARG)` is expanded when the meta-grammar is processed to incorporate the defined number of functions and their arguments. Grammar 5.6 and 5.7 and 5.8 and 5.9.

```
<expr> ::= ( <op> <expr> <expr> ) | <var>
<op>   ::= +|-|*|/
<var>  ::= x|(GeneralRandomConstant)
```

Grammar 5.6: Symbolic Regression meta-grammar, *std* - For the standard GE grammar

```
<prog>     ::= <expr> " (define adf0 (lambda (x) ("<adfexpr1>") ) )"
<expr>     ::= ( <op> <expr> <expr> ) | <var> | (adf0 <expr> )
<op>       ::= +|-|*|/
<var>      ::= x|(GeneralRandomConstant)
<adfexpr1> ::= <op> <adfexpr> <adfexpr>
<adfexpr>  ::= ( <op> <adfexpr> <adfexpr> ) | <adfvar>
<adfvar>   ::= x|(GeneralRandomConstant)
```

Grammar 5.7: Symbolic Regression meta-grammar, *adf* - The GE grammar can define one function with one argument.

5.3.6 Results - Symbolic Regression

For the symbolic regression problems there was no benefit from ADFs, results are in Tab. 5.4

A plot of symbolic regression problems Eq. (5.1) is shown in Fig.5.8. The Fig. 5.8(a) shows how the behaviour after a longer run.

The average best fitness of the last generation is for 50 generations $\overline{std} = 0.129$, $\overline{adf} = 0.215$, $\overline{adf_{dyn}} = 0.333$ and $\overline{adf_{mg}} = 0.714$. The standard GE performs significantly better

```
<prog>     ::= <expr> " "<adfs>
<expr>     ::= ( <op> <expr> <expr> ) | <var> | (adf* <expr> )
<op>       ::= +|-|*|/
<var>      ::= x|(GeneralRandomConstant)
<adfs>     ::= <adf_def> | <adf_def> <adfs>
<adf_def>  ::= "(define adf*(lambda (x) ("<adfexpr1>") ) )"
<adfexpr1> ::= <op> <adfexpr> <adfexpr>
<adfexpr>  ::= ( <op> <adfexpr> <adfexpr> ) | <adfvar>
<adfvar>   ::= x|(GeneralRandomConstant)
```

Grammar 5.8: Symbolic Regression meta-grammar, adf_{dyn} - The GE grammar for creating any number of functions. Each function takes one argument.

```
<g> ::=
       "<prog>   ::= <expr> "
       <adfs>
       "<expr>   ::= ( <op> <expr> <expr> ) | <var> | adf* (ADFARG)"
       "<adfarg> ::= ( <op> <expr> <expr> ) | <var>"
       "<op>     ::= +|-|*|/"
       "<var>    ::= x|(GeneralRandomConstant)"
<adfs>     ::= <adf_def>split<adfs> | <adf_def>
<adf_def>  ::= "(define adf* (lambda ("<adfparam>") ("<adfexpr1>") ) )"
<adfparam> ::= ADFPARAM <adfparam> | ADFPARAM
<adfexpr1> ::= <adfop> <adfexpr> <adfexpr>
<adfexpr>  ::= ( <adfop> <adfexpr> <adfexpr> ) | <adfvar>
<adfvar>   ::= ADFUSE|(GeneralRandomConstant)
<adfop>    ::= +|-|*|/
```

Grammar 5.9: Symbolic Regression meta-grammars, adf_{mg} - meta-GE grammar multiple parameters.

for the against adf_{dyn} and adf_{mg}.

A plot of Eq. (5.2) is shown in Fig. 5.9 The average best fitness of the last generation is for 50 generations $\overline{std} = 0.777$, $\overline{adf} = 1.262$, $\overline{adf_{dyn}} = 1.349$ and $\overline{adf_{mg}} = 1.590$. Standard GE is significantly better.

Plots of dynamic problems, switching between Eq. (5.1) and Eq. (5.2) and increasing the polynomial by one term every 10 generations Eq. (5.3) are shown in Fig.5.10. The average best fitness of the last generation is when switching between Eq. (5.1) and Eq. (5.2) $\overline{std} = 3.655$, $\overline{adf} = 2.337$, $\overline{adf_{dyn}} = 3.335$ and $\overline{adf_{mg}} = 2.630.25$. There is no significant difference between any of the grammars. The average best fitness of the last generation for Eq. (5.3) $\overline{std} = 0.089$, $\overline{adf} = 0.101$, $\overline{adf_{dyn}} = 0.153$ and $\overline{adf_{mg}} = 0.144$, here there are no significant differences. In conclusion, for the problems used here Symbolic Regression does not show any clear benefits from the incorporation of ADFs.

5.3.7 Discussion

The results showed that there were benefits to be had by using ADFs on the Ant Trails, but we could not distinguish any difference between the various grammars and algorithms.

Tab. 5.4: p-values for the grammars compared to the standard grammar on the different problems, the average best fitness and standard deviation at the last generation is shown next to the grammar. Underlined indicate a significant p-value.

$x + x^2 + x^3 + x^4 + x^5$

Grammar	**Std**(0.129 ± 0.202)
$adf(0.215 \pm 0.348)$	$1.03e-01$
$adf_{dyn}(0.333 \pm 0.549)$	$7.99e-03$
$adf_{mg}(0.714 \pm 0.719)$	$7.18e-08$

$x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^{10}$

Grammar	**Std**(0.777 ± 0.592)
$adf(1.262 \pm 0.574)$	$2.08e-03$
$adf_{dyn}(1.349 \pm 0.919)$	$5.72e-03$
$adf_{mg}(1.590 \pm 1.661)$	$1.33e-02$

Altering

Grammar	**Std**(3.655 ± 1.941)
$adf(4.241 \pm 2.337)$	$2.95e-01$
$adf_{dyn}(4.890 \pm 3.335)$	$8.50e-02$
$adf_{mg}(4.783 \pm 2.630)$	$6.38e-02$

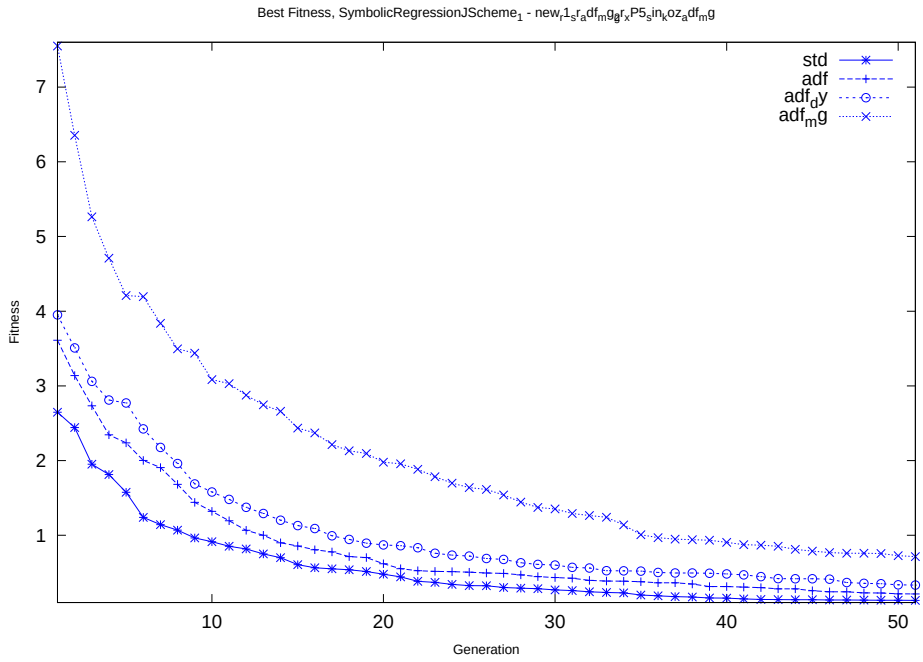
Increasing

Grammar	**Std**(0.089 ± 0.341)
$adf(0.101 \pm 0.369)$	$9.03e-01$
$adf_{dyn}(0.153 \pm 0.341)$	$4.72e-01$
$adf_{mg}(0.144 \pm 0.464)$	$6.04e-01$

Thus it seems that the most important factor is the use of a module, the number of modules is not important.

Example solutions of the San Mateo Trail see Appendix A. For the Symbolic Regression problems investigated it seems there were no benefits to be had by using ADFs. This could be dependent on several factors, the parameters used for the GE run, the fitness function and the points chosen for function comparison. Koza [81] finds some symbolic regression problems of a certain form do not benefit from the use of ADFs, with the solution being found faster without ADFs.

The performance of the grammars differs for all the problems, it seems as if both the type and the size of the problem are very influential. Harper and Blair [52] argued that the meta-grammar approach would benefit from structure-preserving operators. Structure preserving operators are outside the scope of this thesis, since our focus is the grammar.



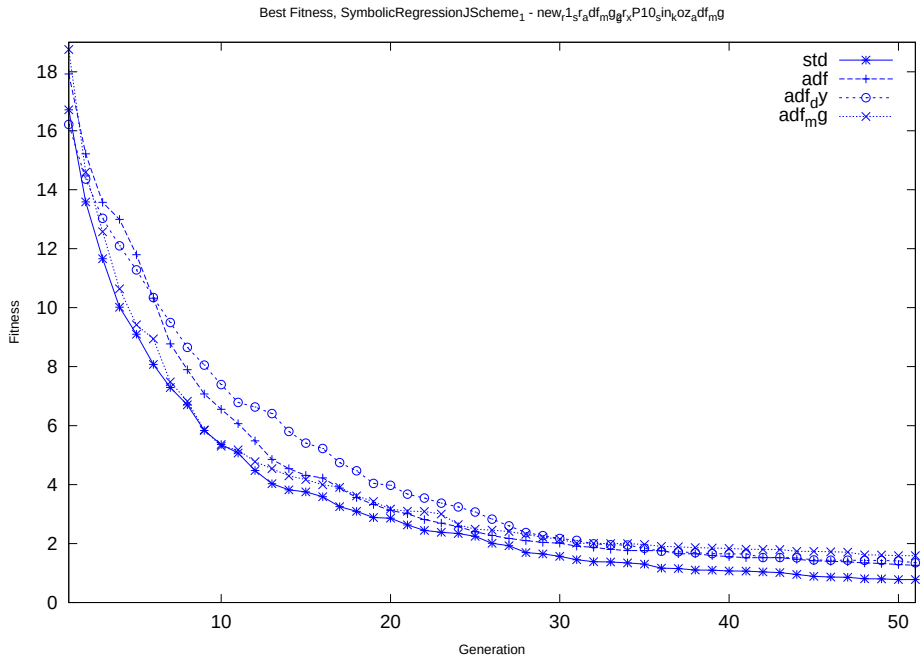
(a) 50 Generations

Fig. 5.8: Plot over generations for Eq. (5.1).

5.4 Summary

This chapter presented an implementation of a meta-grammar GE for capturing modularity by using dynamic definition of modules with fixed module signatures. Automatically Defined Functions are a fundamental tool adopted in GP to allow problem decomposition and leverage modules in order to improve scalability to larger problems. We examined a number of function representations using GE. The problems investigated are variants of the ant trail, static and dynamic Symbolic Regression instances. For the problems examined we find that irrespective of the function representation, the presence of Automatically Defined Functions alone is sufficient to significantly improve performance for problems that are complex enough to justify their use.

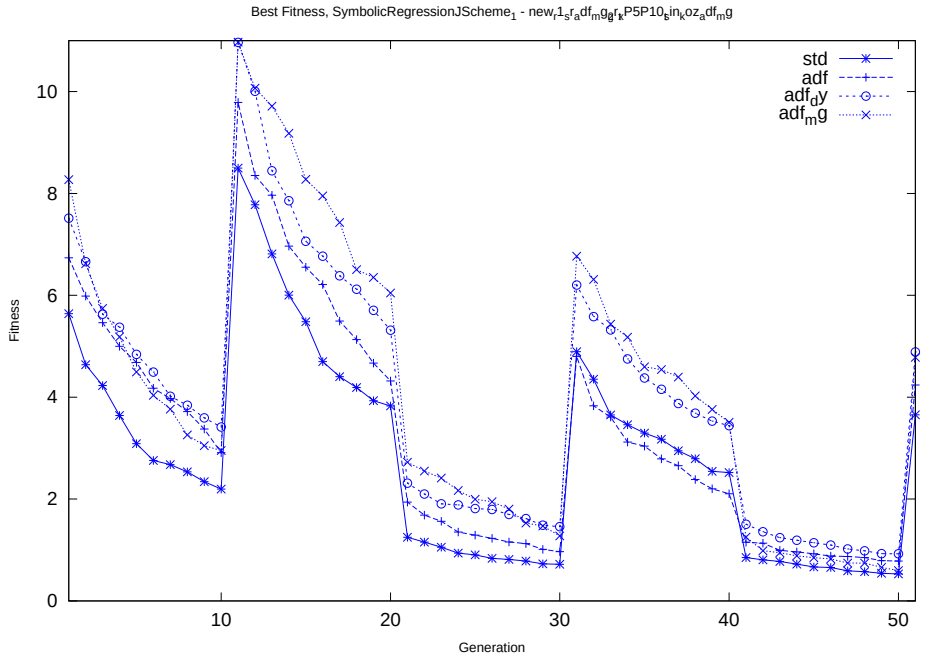
We will extend the study of meta-grammars in the following chapters, since there is some benefit to be had by using meta-grammars to capture modules. In Chapter 6 the meta-grammar approach to capture modules in the problem that can be reused later is



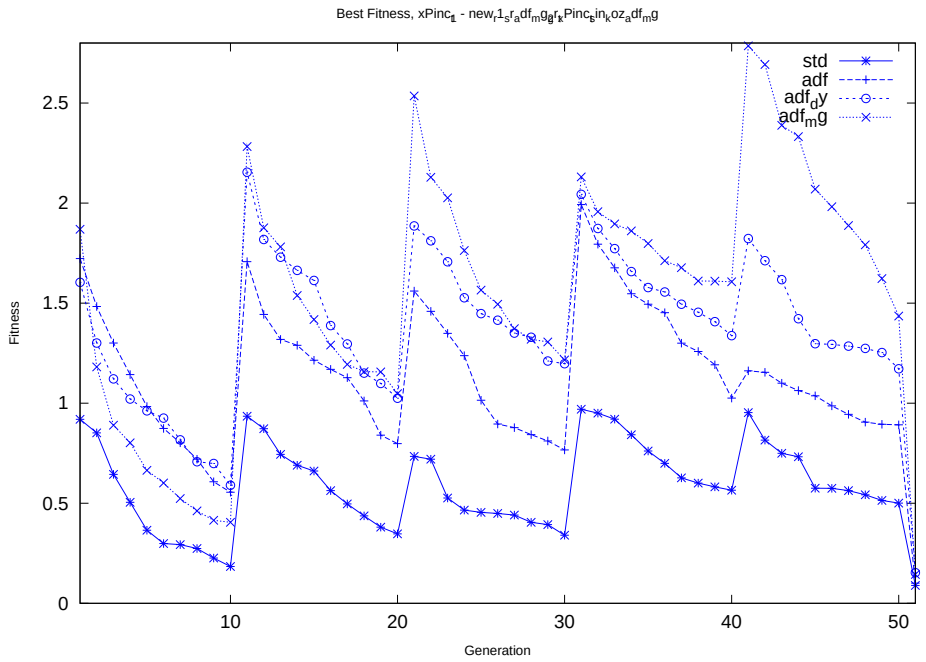
(a) 50 Generations

Fig. 5.9: Plot over generations for Eq. (5.2).

further studied. The ability to reuse modules will facilitate solving problems of different scales which contain repetitive modules.



(a) Eq. (5.1) and Eq. (5.2) switching



(b) Eq. (5.3)

Fig. 5.10: Plot over generations for dynamic functions with a period of 10. Left periodic switching between Eq. (5.1) and Eq. (5.2) and right Eq. (5.3).

Chapter 6

Meta-Grammar for Genetic Algorithms - Scalability

This chapter studies an implementation of a meta-grammar for GE, called the meta-grammar Genetic Algorithm (mGGA). The goal of this meta-grammar approach is scalability, which is facilitated by capturing modules in the problem that can be reused later. This property is beneficial when solving problems of different scales which can be broken up into repetitive modules; by allowing the reuse of modules the representation can become more compact. By using a simple problem the effects of the meta-grammar approach can be analyzed more clearly. This chapter reports the scalability and behavior of the mGGA on the Checkerboard, as investigated in Hemberg et al. [60].

Section 6.1 presents the meta-grammar algorithm, called the mGGA, used in the experiments. The setup and results regarding experiments on the scalability of the mGGA are given in Section 6.2. Finally, the chapter comes to an end with a summary in Section 6.3.

6.1 Meta-Grammar Genetic Algorithm

This section describes the mGGA algorithm, some examples of how it captures modularity, as well as previous research regarding Grammatical Evolution (GE) and meta-grammars.

The purpose of the mGGA is to study modularity and more specifically to capture regularities in the problem by using building block structures in the grammar, and thereby increasing its scalability. The mGGA approach to modularity adopts principles from GP, variable length representations, and uses the declarative bias of the grammar to find structures able to represent a fixed-length problem efficiently. This differs from the fixed-length GA approach to modularity which has examined links between the variables in the representation. An example of previous work on grammars and GAs is GAUGE, see Section 3.1.4.

The grammar-based GP approach upon which this study is based is the GE^2 algorithm [114], see Section 5.2 on page 76, which is in turn based on the GE algorithm [115]. In a meta-grammar GE algorithm the input grammar is used to specify the construction of another syntactically correct grammar. In the mGGA [111] the meta-grammar approach was shown as an alternative binary string GA and the use of modules improved the performance of the mGGA. The generated grammar is then used to generate a solution, this process is illustrated in Fig. 5.2 on page 77. In this implementation the mGGA is allowed to evolve bias towards different building block structures of varying sizes and content.

Crossover in the mGGA operates between homologous chromosomes, with the meta-grammar chromosome from the first parent recombining with the meta-grammar chromosome from the second parent, the same occurs for the solution chromosomes. In order for evolution to be successful it must co-evolve both the meta-grammar and the structure of solutions based on the evolved meta-grammar, and as such the search space is larger than in standard GE.

In Section 6.1.1 the description of the mGGA will focus on how a grammar might be used to encode binary strings, and finally on how a meta-grammar can represent a binary-string grammar.

6.1.1 Grammars for Bit Strings

A simple BNF grammar for a fixed-length (eight bits in the following examples) binary string individual of a GA, called GEGA is shown in Grammar 6.1. In the generative

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

Grammar 6.1: GEGA grammar for producing a bitstring of length eight.

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

Grammar 6.2: GE bit string grammar with building block structures `<bbk4>`, `<bbk2>`. I.e. the reuse of groups of bits (building block structures) into a more compact representation of the bit string.

grammar each bit position (denoted as the non-terminal `<bit>`) can become either of the Boolean terminal values, `0` or `1`. A standard variable-length GE individual can then be allowed to specify what each bit value will be by selecting the appropriate `<bit>` production rule for each position in the `<bitstring>`.

In order to have useful recombinations of building blocks the representation must be such that the building blocks exist [2]. The grammar in Grammar 6.1 can be extended to incorporate the reuse of groups of bits (building block structures) into a more compact representation of the bit string, called GEGABB. In this grammar example, Grammar 6.2, all building block structures that are multiples of two are provided. This allow a stronger declarative bias towards these structures.

The grammars in Grammar 6.1 and Grammar 6.2 are static. Grammar 6.2 can only allow one building block structure of size four and one of size two. The algorithm can gain more freedom by allowing the search the potential to uncover a number of building block structures of any size from which a GE individual could choose, for an example see Grammar 6.3. This would facilitate the application of such a grammar-based GA to:

```
<bitstring> ::= <bbk4><bbk4>
              | <bbk2><bbk2><bbk2><bbk2>
              | <bbk1><bbk1><bbk1t><bbk1><bbk1><bbk1><bbk1><bbk1>
<bbk4> ::= <bbk4_0>
         | <bbk4_1>
<bbk4_0> ::= <bit><bit><bit>0
<bbk4_1> ::= <bit><bit><bit><bit>
<bbk2> ::= <bbk2_0>
         | <bbk_1>
<bbk2_0> ::= <bit>0
<bbk2_1> ::= <bit><bit>
<bbk1> ::= <bit>
<bit> ::= 1
        | 0
```

Grammar 6.3: GE bit string grammar with choices between building block structures. Rules differing from Grammar 6.4 are shown in *italics*

- problems with more than one building block structure type for each size
- the search for one building block structure while keeping a *reasonable* temporary solution
- the ability to act as a building block structure memory and to switch between different building block structures if the environment changes

The derivation tree generated from Grammar 6.4 is such that the rule `<g>` has five branches, which generate number of repetitions, the number of different building block structures and content and the `<bit>` bias, e.g. see Fig. 5.3. Each building block structure branch evolves the building block structure, where the `<bit>` is connected to the last branch, and the availability of building block structures is determined by the first branch. Exchange of these branches provide transfer of material between indviduals. In Section 6.1.2 an example of the mGGA is provided along with grammars for generating bit strings and capturing building block structures.

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

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

Grammar 6.4: Meta-grammar with building block structures example (mGGABB).

6.1.2 Examples of mGGA Grammars

An example of a meta-grammar for an individual generating an eight bit string called mGGABB, is given in Grammar 6.4. The ability to specify which building block structures to allow in the solution grammar comes from the use of recursion in `<reps>`, whereas the recursion in `<val>` enables changes of the Boolean terminal bias.

In this case the grammar specifies the construction of another generative bit string grammar. The subsequent bit string grammar that can be produced from the mGGABB (Grammar 6.4) is restricted in such a way that it can contain building block structures of size two and four. Some of the bits of the building block structures can be fully specified as a Boolean value or may be left as unfilled for the second step in the mapping process. An example of bit string grammar produced from the mGGABB meta-grammar is shown in Grammar 6.5.

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

Grammar 6.5: Example of a solution grammar produced by mGGABB (Grammar 6.4)

The entire mapping process is illustrated in Fig. 6.1 on page 101 by the generation of a binary string of size four, using a grammar similar to the one in Grammar 6.4.

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

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

Grammar 6.6: Meta-grammar with multiple building blocks (mGGAMBB). Rules differing from Grammar 6.4 are shown in *italics*

To allow the creation of multiple building block structures of different sizes, the following grammar could be adopted, called mGGAMBB (again shown for bit strings of length eight), as shown in Grammar 6.6. The multiple building block structures in the solution grammar are enabled by the use of recursive rules for `<bbk4t>`, `<bbk2t>` and `<bbk1t>`.

Grammar 6.7 shows an example of bit string grammar generated by mGGAMBB.

In the example of a bit string grammar, the solution grammar in Grammar 6.7 there are five possible forms that a `<bitstring>` can take on, with two possible choices for building block structures of size four and one, and three choices for building block structures of size

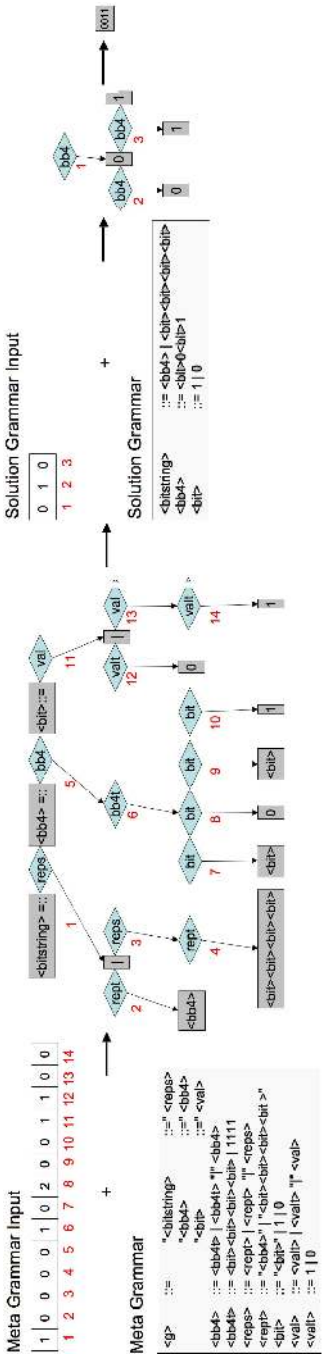


Fig. 6.1: An example of the mapping of a meta-grammar. Diamonds are non- terminal symbols and rectangles are terminal symbols. The numbers by the arrows are used to denote which input chooses the production from the rule.

```
<bitstring> ::= <bit>11<bit>00<bit><bit>
              | <bb2><bb2><bb2><bb2>
              | 11011101
              | <bb4><bb4>
              | <bb4><bb4>
<bb4> ::= <bit>11<bit>
        | 000<bit>
<bb2> ::= 11
        | 00
        | <bit>1
<bb1> ::= 0
        | 0
<bit> ::= 1
        | 0
        | 1
        | 1
```

Grammar 6.7: Example of a solution grammar from mGGAMBB (Grammar 6.6). In `<bit>` a bias towards `1` can be seen.

two. The rule for generating a `<bit>` has four possible outcomes with a clear bias towards a `<bit>` becoming a `1` with a probability of 0.75 (three out of the four production choices will result in the terminal `1`) .

To conclude, modularity exists in both the mGGABB and mGGAMBB grammars, in their ability to specify the different sizes and contents (or partial contents) of building block structures through their incorporation into the solution grammar. The building block structures can then be reused repeatedly in the generation of the phenotype. This can facilitate the search in some regular structures and make the mGGA suitable for finding repeating patterns. The declarative bias in the grammar and properties like these would make the mGGA suitable for problems of increasing size, since it has a representation which is able to compress the solution.

The following sections describe a series of experiments involving the mGGA. In Section 6.2 a scalability study is outlined, comparing the mGGA to the Modular Genetic Algorithm (MGA).

6.2 Scalability of the mGGA

This section studies the performance of the mGGA on problems of increasing size. First we define regularity in Section 6.2.1. Then an approach tailored to modularity and regularity based on a GA, the MGA [40], which is also similar to the mGGA in its approach to modules, is presented in Section 6.2.2. We also present a comparison of the mGGA and MGA for different sizes of the Checkerboard problem and some noisy variants of it. The benchmark Checkerboard problem is used to test the ability of the mGGA to capture modules. Before detailing the experimental design and setup the Checkerboard problem will be introduced.

6.2.1 Regularity

Here we define regularity as reuse of modules. This can reduce the information needed to describe the design [88]. The connection between modularity and regularity leads to enforcement of regularity to create modularity with different representations. Instead of invoking regularity to enhance modularity, partial specification can be used, with explicit definition of modules where the unit of selection is a partial solution that is represented independently and the module is evaluated in the context of other modules. De Jong and Thierens [29] introduces an algorithm for exploitation of modularity, hierarchy, and reuse. That study indicates that the simultaneous exploitation of hierarchy and repetition will require both position-specific module testing and position-independent module use.

Garibay et al. [40] approach regularities in GAs by using a run length encoding, i.e., one symbol is used to indicate one or zero and a number is used to indicate multiple repetitions of ones or zeros. The MGA introduced by Garibay et al. [40] was shown to significantly outperform a standard GA on a scalable problem with regularities. Here we introduce a meta-grammar approach that is able to capture regularities that are not only repetitions of single pre-defined symbols.

6.2.2 Modular Genetic Algorithm

The MGA can be described as an encoding where more than one symbol is replaced by a digit indicating the number of repetitions, the aim was to create an algorithm for automatic module discovery.

The genome of an MGA individual is a section of a vector bundle of genes, $g = (g_1, \ldots, g_i)$, and each gene is a tuple, $(n_i, f_i())$, where n_i is the number of times that some function, $f_i()$ is repeated. This gives when expanded

$$\begin{aligned} g_i &= \langle n_i, f_i() \rangle \\ &= f_{i_1}(), \ldots, f_{i_j}(), 0 \leq j \leq n_i \end{aligned}$$

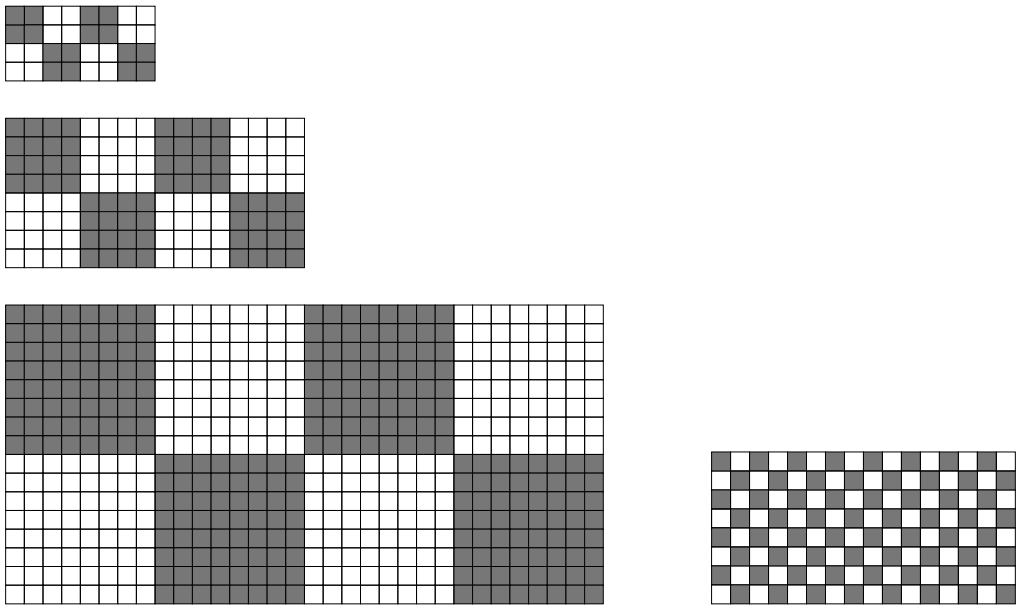
For example, if there was a choice for $f_i \in \{e(), z()\}$ and $0 \leq n_i \leq N$, where function, $e() = 1$ always returned the value 1 when called, and another $z() = 0$ returned the value 0 , this would be a representation that can generate binary strings. A sample individual comprised of three genes might look like: $g = ((2, z()), (4, e()), (2, z()))$, which would output the binary string 00111100. The operators for the MGA are mutation, which changes both values of the gene tuple, g_i. Crossover acts only between genes, splitting genes related to crossover is not allowed. The initialization is done using a uniform probability for the tuple values.

6.2.3 Checkerboard Problem

Garibay et al. [40] identified a lack of suitable problems for the study of regularity and modularity in GAs and thus proposed the Checkerboard pattern discovery problem. This is a generalized OneMax pattern matching problem [39], where a pattern of states or colors is imposed upon a two dimensional grid called the Checkerboard. There are two possible states adopted for each square on the grid, i.e. white or black, which can be represented by the values 0 and 1. Each candidate solution tries to recapture the pattern contained in the target Checkerboard, with fitness being the Hamming distance between the Checkerboard pattern and the candidate solution.

In this study the fitness is normalized to the range 0.0 to 1.0, and the problem is to minimize the fitness, i.e. 0.0 is the best possible fitness, a complete match. It is easy to scale the problem in terms of its complexity, modularity and regularity by increasing the size of the Checkerboard, the number of patterns, and by changing the number of components in each pattern respectively. The size of the search space is 2^l, where l is the number of squares on the board. Instances of the Checkerboard problem which are adopted in this study and in Garibay et al. [40] are presented in Fig. 6.2(a), which illustrates scaled-up versions, Cb_{32}, Cb_{128_0} and Cb_{512}. Another problem instance introduced and tackled in this study is the Cb_{128_1} Checkerboard pattern, also shown in Fig. 6.2(b).

A third set of problem instances is also examined, which adds noise to the state of each



(a) Cb_{32}, Cb_{128_0} and Cb_{512} (b) Cb_{128_1}

Fig. 6.2: The original Checkerboard-pattern matching problem instances in Fig. 6.2(a) (from the top Cb_{32}, Cb_{128_0} and Cb_{512}). Fig. 6.2(b) Cb_{128_1} shows a new Checkerboard-pattern matching problem instance with a more fine-grained regularity.

square. These noisy instances are an extension of the original Checkerboard problem. The noise is implemented by randomly flipping the state of a square with a uniform probability for the patterns presented in Fig. 6.2(a). The addition of noise to the regular patterns makes it more challenging to uncover the underlying patterns and thus adds an additional element of real-world interest to this benchmark problem. The amount of noise can easily be tuned by altering the probability for changing the state of a square. According to Rohlfshagen et al. [134] it can be seen as an XOR dynamic function.

6.2.4 mGGA on the Checkerboard

The aim is to see if the mGGA has a better performance than the MGA on the Checkerboard, 30 runs were performed. The performance is how well the the solution matches the target, ϕ_{mGGA} is the performance for the mGGA and ϕ_{MGA} is the performance for the MGA. The number of fitness evaluations were the same for both mGGA and MGA.

Tab. 6.1: Parameters for the GE algorithm

Parameter	Values
Checkerboard size	32, 128, 512
Fixed chromosome size	90, 320, 1300
Initialization	Random
Selection operation	Tournament
Tournament size	3
Replacement	Rank replacement
Max wraps	1
Population Size	1000
Generations	500, 1000, 2000
Crossover type	Fixed one point
Crossover probability	0.7 (Both chromosomes)
Mutation type	Integer Flip
Mutation probability	0.001 (Both chromosomes)

A major obstacle is that only the averages for the MGA are reported [40], and not the standard deviations (only visually with 95% confidence interval), this makes it difficult to perform a test for comparison.

When conducting the experiments both chromosomes were variable-length vectors of integers (4 byte integers) and had the same initial length. Rank replacement is adopted with a constant population size, where the new children are pooled with the current population, ranked, and the worst individuals are removed. The mutation was done by uniformly choosing a new integer value for the mutated codon. For crossover, which is homologous, a one-point crossover with the same crossover point is used for both parents. The settings in Tab. 6.1 were adopted for the experiments and they were implemented using GEVA [118].

To solve the Checkerboard a meta-grammar was created and to allow the creation of multiple building block structures of different sizes the mGGAMBB was used (Grammar 6.6). For the grammars the building block structures were specified as follows: when ss_i, the number of consecutive ones or zeroes, a building block structure is created for $2^j, j \leq log_2(N/2), j \in \mathbb{N}$, N is the total number of squares on the Checkerboard.

Tab. 6.2: Performance changes for the mGGA on the standard non-noisy problem instances (the Cb_{512} has performed better).

Performance drop (% of fitness decrease)		
Complexity increase	MGA	mGGA
from Cb_{32} to Cb_{128_0}	3.68%	0.02%
from Cb_{32} to Cb_{512}	11.38%	-0.1%

Tab. 6.3: Performance values for the mGGA on the non-noisy problem instances. Average values and success rate are for 30 runs. Success rate is the proportion of successfull solutions over the runs. The value in parenthesis is the fitness for random search for 10^6 tries.

Problem	**Best fitness±Std**	**Success Rate**	**Random Search**
Cb_{32}	0.01979 ± 0.05138	0.87	0.11
Cb_{128_0}	0.01953 ± 0.05158	0.83	0.16
Cb_{128_1}	0	1.00	0.29
Cb_{512}	0.01875 ± 0.15735	0.90	0.42

Results

The results, with the percentage gains in performance and fitness statistics are reported in Tab. 6.2 and 6.3 respectively. The average best fitness after 500 generations is 0.01979 for Cb_{32} and after a 1000 generations 0.01953 for Cb_{128_0}. The difference in fitness between the two instances is 0.00026. The average best fitness after 400 generations for Cb_{512} is 0.01875. Difference between Cb_{32} and Cb_{512} is 0.00104. It is clear that as the problem instances increase in complexity there are economies of scale to be achieved, with the relative performance of the mGGA improving with each jump in problem size. The mGGA beats random search on the non-noisy problems.

An additional problem instance, Cb_{128_1} as portrayed in Fig. 6.2(b) was examined with the same parameters as previously. The results are presented in Fig. 6.3(a), for this instance the pattern is as such much more fine-grained than the ones examined earlier. This makes it difficult for the MGA to efficiently represent a solution to this problem instance due to the nature of the pattern. Effectively each square's state must be specified individually. However, this is not the case with the mGGA which can encode effectively and parametrize the evolved modules to specify multiple square states with different values.

Tab. 6.4: Statistics for performance of the mGGA on the Noisy Checkerboard instances for Cb_{128_0}. Success rate is the proportion of successfull solutions over the runs.

Noise level	Best fitness±Std	Success Rate
$p_n = 0$	0.0195 ± 0.0520	0.83
$p_n = 0.025$	0.0198 ± 0.0447	0.80
$p_n = 0.05$	0.0664 ± 0.1109	0.73
$p_n = 0.075$	0.0841 ± 0.0990	0.43

6.2.5 mGGA Performance under Noisy Conditions

In order to gain some preliminary insight into the performance of the mGGA in a more realistic real-world setting, experimental runs incorporating noise into the target patterns were conducted. This was achieved by flipping each bit in the target pattern with a uniform probability, p_n. Runs were conducted using the same parameters as previously described for noise probabilities $p_n = 0.05, 0.075$ for the Cb_{128_0}. The results are presented in Tab. 6.4 and there it is possible to see that the performance decreases when the noise is increased.

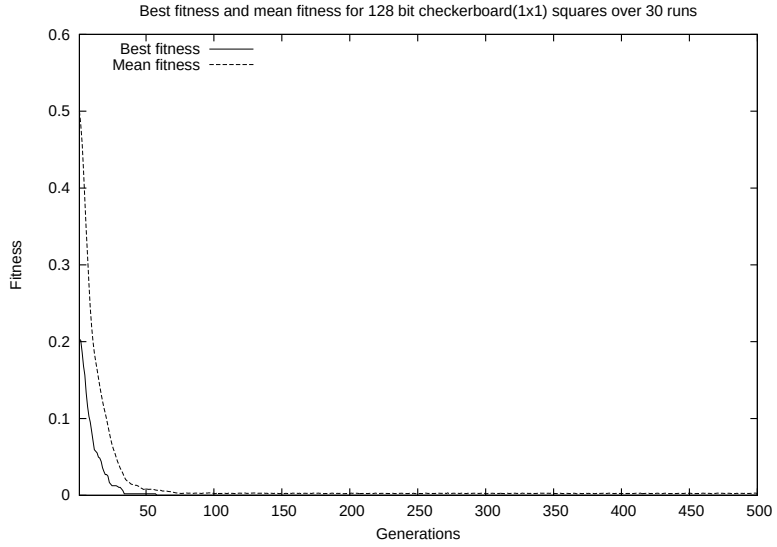
As can be expected, the addition of noise reduced the algorithm's performance on average. However, on inspection of individual runs it was seen that this performance drop was manifest in an increased, but still small, number of runs. The faild runs converged prematurely to a suboptimal solution, with Fig. 6.3(b) showing the results for the 0.05 noise level. This indicates that the population may be converging too quickly in the early stages of the algorithm, loosing whatever diversity was present in the initial population. It is possible that through adjusting parameters of the EA better results could be achieved.

6.3 Summary

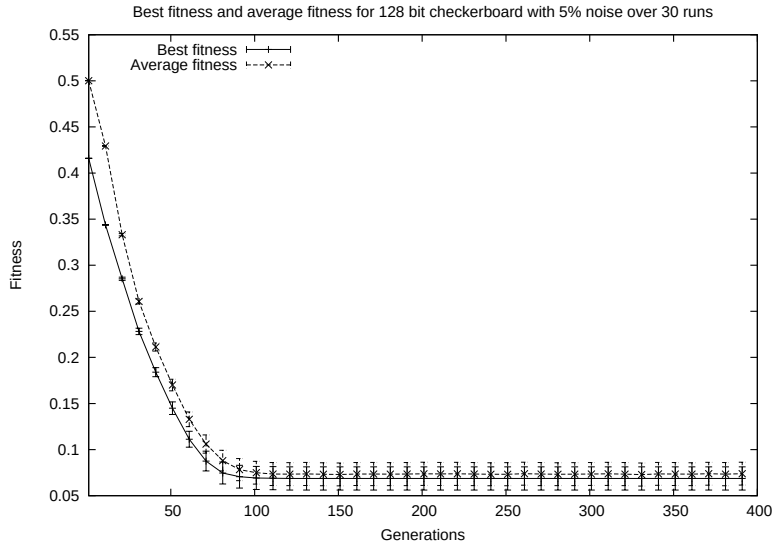
Genetic Programming has the ability to scale to problems of increasing difficulty since it is possible to capture regularities that exist in a problem environment by decomposition of the problem into a hierarchy of modules. In this chapter we considered the adoption of a decompositional strategy for a fixed length problem. By adopting a modular representation in a fixed length problem we can make efficiency gains that enable scaling

to problems of increasing size. We presented a comparison of two modular Genetic Algorithms, one of which is a grammatical Genetic Programming algorithm, the meta-grammar Genetic Algorithm (mGGA), which generates binary string sentences. A number of problem instances are tackled which extend the Checkerboard problem by introducing different kinds of regularity and noise. The results demonstrate some limitations of the modular GA (MGA) representation and how the mGGA can overcome these. The mGGA shows improved performance when scaling the problems when compared to the MGA.

The operations used for the mGGA are furthered studied in Chapter 7. The exploration is on the impact on performance by using operations to diversify the two chromosomes.



(a) $Cb_{128_1}, p_n = 0$



(b) $Cb_{128_0}, p_n = 0.05$

Fig. 6.3: A graph for the mGGA for Cb_{128_1} Checkerboard (1X1) is shown in 6.3(a), and the standard Cb_{128_0} instance with $p_n = 0.05$ in 6.3(b).

Chapter 7

Altering Search Rates of the Meta- and Solution Grammars in the mGGA

In Chapter 5 and Chapter 6 we have investigated the ability of a meta-grammar approach to capture modules, as well as the ability to handle problems of increasing size. In order to investigate the ability of the mGGA to identify and use modules its operators are studied more closely; this was first investigated in a paper by Hemberg et al. [59]. In this chapter two approaches to altering the rate of exploration of the solution grammars are examined in order to understand mixing operators in the mGGA. The aim is to understand how to guide the meta-grammar search in order to improve the performance of the mGGA. The first approach adopts an implicit sampling by using different rates of mutation for the meta-grammar and the solution grammar. With a lower mutation rate on the meta-grammar chromosome the expected number of changes, making it possible to sample solution grammars more frequently, see Fig. 7.1. The second approach explicitly generates more than one sample from each solution grammar, see Fig. 7.2.

For the problem instances examined neither approach conclusively improved the performance of the meta-grammar approach to GE. Although, for the different mutation rates a

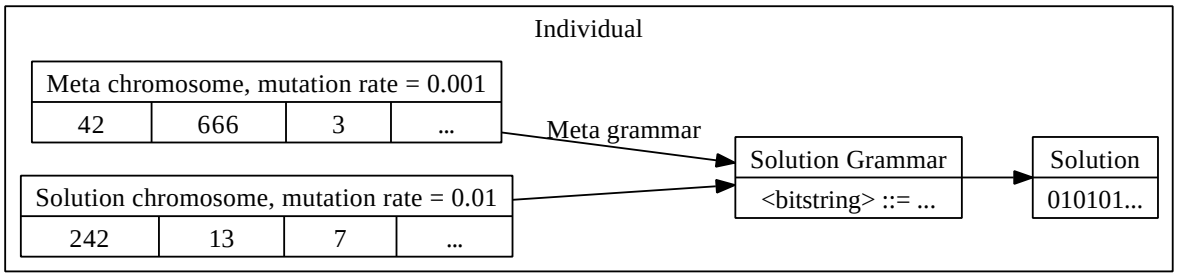


Fig. 7.1: Implicit sampling of a meta-grammar GE

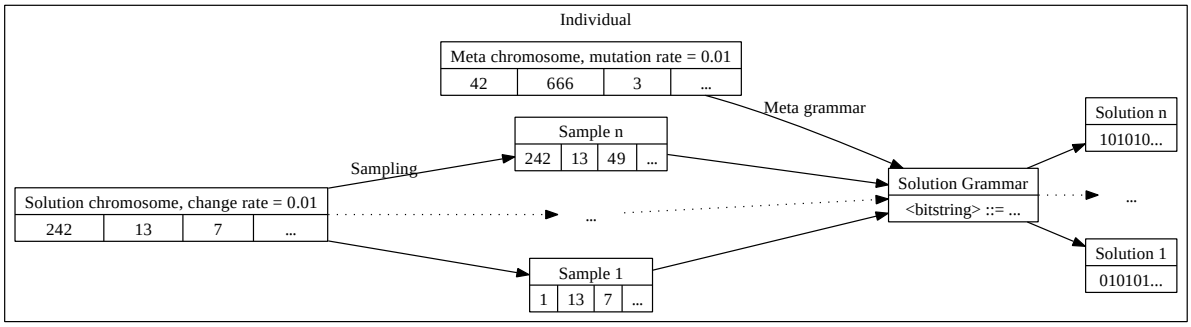


Fig. 7.2: Explicit sampling of a meta-grammar GE

lower mutation rate improved performance on some instances. The majority of the evolutionary search is focused on the generation of the solution grammars leaving the candidate solutions often almost hard-coding the solution, i.e. the search space of the solution grammar was non-existent or very small.

Section 7.1 describes the implicit sampling approach of using different mutation rates for the meta- and the solution chromosome. In Section 7.2 the explicit sampling of the meta-chromosome is presented. Results are further discussed in Section 7.3. The chapter is concluded with a summary in Section 7.4.

7.1 Different Mutation Rates

The first experiment tests if there is any improvement in the performance of the mGGA by having different mutation rates for the two chromosomes. An argument can be made that

if the meta-grammar evolved at a slower pace then exploration of the solution grammar would be possible, thus creating a version of a meta-grammar local search. Keller and Banzhaf [78] investigated the evolution of the genetic code for GP with the aim of enhancing the search process. Their hypothesis was that since the genetic code in nature has evolved with organisms, evolution of the code for artificial evolution would aid the search. The aim is to allow the individuals' code to be biased to the problem, something which can be beneficial in dynamic environments.

Hypothesis

The performance is measured as the average number of fitness evaluations required for 30 runs to solve an instance of the Checkerboard. μ is the performance for the mutation being the same for both chromosomes. The performance for the chromosome having different mutation rates is referred to as μ_0. For each instance the following hypothesis is stated:

H_0**:** Equal mutation rate for both chromosomes has the same performance than a lower mutation rate for the first chromosome, i.e. $\mu = \mu_0$

H_1**:** Equal mutation rates show a different performance if compared to a low mutation rate for the first chromosome, i.e. $\mu \neq \mu_0$

α**:** The significance level of the test is 0.05.

7.1.1 Setup

In the experiments the population size was determined by finding a value within 10% of where 30 runs are successful for a maximum of 800 iterations. The meta-grammar building block structure sizes for the Checkerboard instance are: Cb_{72} has $32, 12, 6, 3, 1$, Cb_{200} has $100, 20, 10, 5, 1$ and Cb_{288} has $144, 24, 12, 6, 3, 1$. The settings in Tab. 7.1 show the alterations of the experimental setup from Tab. 6.1.

Tab. 7.1: Parameters for the GE algorithm in Section 7

Parameter	Values
Checkerboard size	32, 72, 128, 200, 288
Fixed chromosome size	90, 210, 300, 580, 800
Generations	800
Crossover probability meta	0.9 (Both chromosomes)
Mutation probability meta	0.001
Mutation probability solution	0.01

7.1.2 Results

The results for the average number of fitness evaluations used to solve each Checkerboard instance are shown in Fig. 7.3. After performing a t-test it was shown that there is a significant decrease in the average number of fitness evaluations required when using a lower meta-chromosome mutation rate for the problem instances Cb_{32} and Cb_{128_0}.

For the instances Cb_{32} and Cb_{128_0} the lower mutation rate on the meta-grammar chromosome has significantly better performance. On Cb_{72} there is no significant difference, while on the Cb_{200} and Cb_{288} the lower mutation rates are significantly worse. Fig. 7.4 shows fitness progression over time.

The conclusion of these experiments is that for some Checkerboard instances it can be beneficial to differentiate the mutation rates for the meta- and solution chromosome, while for others it can be detrimental. By applying a lower mutation probability for the meta-chromosome the solution grammars can be explored further and would not be disrupted too often by mutation events.

7.2 Sampling Each Solution Grammar n Times

In this experiment, instead of changing mutation rates, i.e. implicitly increasing the frequency of the exploration of the solution grammar, an explicit increase in sampling of the solution grammars is used. This is achieved by randomly mutating the solution chromosome which is used to construct sentences from the evolved solution grammar. For each

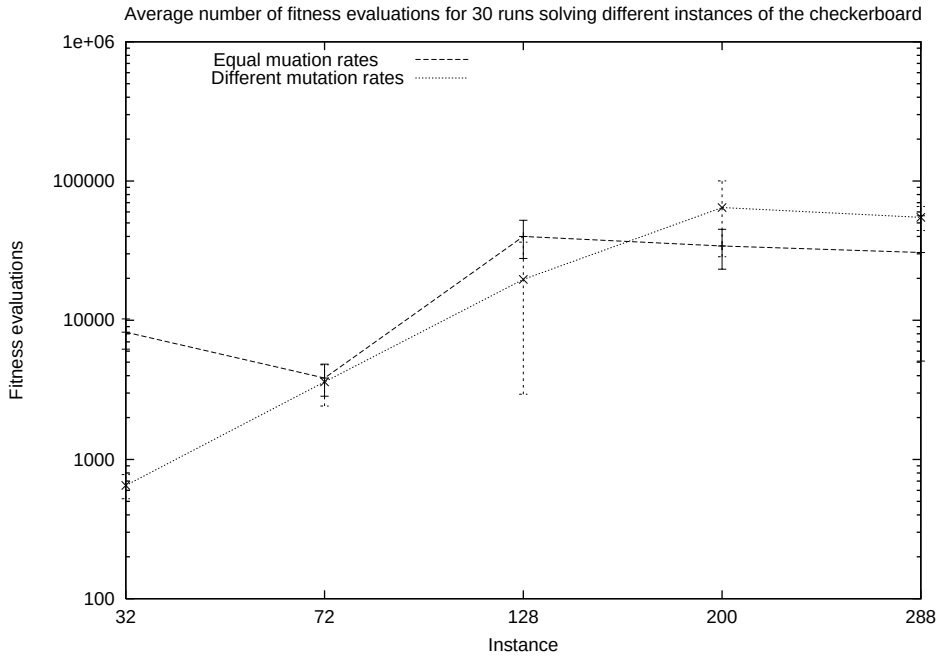


Fig. 7.3: On the x-axis are the problem instances, indicated by the total number of bits, and on the y-axis the number of fitness evaluations (log-scale).

solution grammar n samples are generated, where $n \in \{2, 5, 10, 20, 60\}$, and the fitness for each of the samples from the solution grammar is evaluated.

Explicit Sampling

The sampling of each solution grammar experiment uses the same settings as given in Tab. 7.1, with the exception that the rate of mutation $p_m = 0.01$ is arbitrarily chosen and used for both the meta-grammar and the solution grammar chromosomes. After initializing the population of size S with uniform probability the following algorithm is performed.

0. **Sample** n individuals by mutating the solution chromosome with probability p_m for each individual. Rank the samples. Add the best sample to the population.

1. **Select** S individuals using tournament selection.

2. **Crossover** the S individuals to create S new individuals.

3. **Mutate** the new individuals.

4. **Replace**, add new individuals to old individuals. Rank and remove the S worst.

Hypothesis

As in Section 7.1 the performance is measured as the average number of fitness evaluations required for 30 runs to solve an instance of the Checkerboard. μ is the performance with only one sample from each chromosome. The performance of the n samples from the generated grammar is referred to as μ_{1U-nS}. To correct for multiple comparisons in multiple hypothesis testing the False Discovery Rate (FDR) [9] is used. The p-values are derived from t-tests between μ and μ_{1U-nS}. For each instance the following hypothesis is stated:

H_0**:** None of the changes in the sampling rate of the generated grammar has significant performance difference compared to using only one sample in any of the experiments, i.e. $\mu_{1U-1S} = \mu_{1U-2S}$ and $\mu_{1U-1S} = \mu_{1U-5S}$ and $\mu_{1U-1S} = \mu_{1U-10S}$ and $\mu_{1U-1S} = \mu_{1U-20S}$ and $\mu_{1U-1S} = \mu_{1U-60S}$.

H_1**:** At least one of the increases in sampling of the generated grammar has difference performance over at least one experiment, i.e. $\mu_{1U-1S} \neq \mu_{1U-2S}$ or $\mu_{1U-1S} \neq \mu_{1U-5S}$ or $\mu_{1U-1S} \neq \mu_{1U-10S}$ or $\mu_{1U-1S} \neq \mu_{1U-20S}$ or $\mu_{1U-1S} \neq \mu_{1U-60S}$.

α**:** The significance level of the FDR is 0.05.

Results

The results for the average number of fitness evaluations to solve each instance for each sampling rate are shown in Fig. 7.5. After having performed a t-test on the fitness evaluations to calculate p-values, a FDR for each Checkerboard size was calculated, all the p-values were lower than 0.01. The FDR gave only one significant test for the decrease in the number of fitness evaluations. This was when sampling a meta-grammar for the Cb_{32}. For the rest there was no performance gain in sampling the meta-grammar more. In

terms of the explicit sampling of the grammars there seemed to be no significant difference compared to not doing explicit sampling.

In Fig.7.6 the progression of fitness during the runs is shown. The shifting of the curves is due to the increase in the fitness evaluations required.

To conclude the experiments for different sampling rates of the mGGA, it can be observed that lowering the mutation rate for the meta-grammar chromosome can sometimes increase performance, while explicit sampling in general had no significant impact on performance. Therefore, in order to improve the meta-grammar search altering the search rates on the chromosomes can be beneficial.

7.3 Discussion

The experiments regarding our implementation of an algorithm with a representation that is able to capture modularity, the mGGA, have yielded some interesting results and raised some questions.

7.3.1 Mutation Rate

There are some interesting results for the rate of mutation experiments in Section 7.1. For two out of the five instances there is a difference in performance. This implies that adopting a slower rate of evolution through a lower mutation rate for the meta-grammar chromosome can improve the performance of meta-grammar GE for the problems investigated. Moreover, it never has worse performance.

The results of experiments with explicit sampling presented in Section 7.2 were initially surprising, as there was an expectation that an explicit increase in sampling of the evolved solution grammars would yield performance gains, but this was shown not to be the case. This says is that most of the search is performed by the first chromosome, since it seems as if there would be no significant overall gain in performance by changing the way the algorithms explore the grammar generated by the meta-grammar. This has also been

```
<bitstring> ::=  <bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4>
<bbk4> ::= 1 1 0 0
         | 0 0 <bit> 1
<bit> ::= 1
```

Grammar 7.1: mGGAMBB example solution grammar from implicit sampling. Meta-grammar example from the meta-grammar Grammar 6.6 (mGGAMBB), different mutation rates. Grammar solving Cb_{32}. The evolved building block structure `<bbk4>` is used in `<bitstring>`.

observed in earlier studies on the meta-grammar GE where Dempsey [31] observed that the evolved solution grammars tended to provide little choice in terms of the number of possible solutions they represented. In many cases the solutions were hard-coded, i.e. the search space of the solution grammar was non-existent, into these evolved grammars, and here the results are similar. These results show that the blocks of useful code evolved in the solution grammar sometimes match the pattern completely. For example in Grammar 7.1 a sample of a Cb_{32} grammar that solves the problem when using equal mutation rates on the chromosomes is shown, with only `<bbk4>` invoking a choice. The evolved building block structure `<bbk4>` is used repeatedly in `<bitstring>`.

An example of a Cb_{32} grammar that solves the problem when taking 10 samples from the grammar is shown in Grammar 7.2. In this example `<bitstring>` and `<bbk4>` lead to the solution. The evolved building block structure `<bbk4>` is reused in `<bitstring>`.

7.3.2 Length Inspection

Given the results of the experiments conducted in Section 7, a more detailed analysis of the meta-grammar approach is required. In particular we wish to better understand the sizes of the evolved solution grammars and the relative amount of search being undertaken on the meta-grammar and solution grammar chromosomes.

Meta-grammars can allow longer solutions due to their compressed form, thus overcoming the length bias of the phenotypes from a standard GE grammar. This can be found in GE by studying the probabilities of choosing productions, e.g. a recursive rule can have

```
<bitstring> ::=  <bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4>
               | <bbk16><bbk16>
               | <bbk2><bbk2><bbk2><bbk2><bbk2><bbk2><bbk2><bbk2>
                 <bbk2><bbk2><bbk2><bbk2><bbk2><bbk2><bbk2><bbk2>
<bbk16> ::= 1 0 1 1 1 0 <bit> 1 <bit> 1 <bit> <bit> 0 0 1 1
          | 1 <bit> 0 0 <bit> 1 0 <bit> 0 <bit> 0 <bit> <bit> <bit> <bit> 1
<bbk4> ::= <bit> 1 0 0
         |  0 0 <bit> 1
         |  1 0 <bit> 0
<bbk2> ::= <bit> 0
         |  0 <bit>
         |  0 <bit>
<bit> ::= 1
```

Grammar 7.2: Meta-grammar example (mGGAMBB), from 10 samples, solving Cb_{32}

a large impact on the size of the phenotype. Another approach to length bias in GE, and this is also in a sense a compact grammar representation, would be to use non-uniform probabilities based on position in derivation and not only on previous choice of production. Another, less general approach is to bias the grammar even further and use a larger and less compact grammar.

The average length of codons used at the end of each run is shown in Fig. 7.7. The number of codons used in the solution grammar chromosome does not increase as much as the number of used chromosomes in the meta-grammar chromosome when the problem sizes increase, and there are a lot more expressed codons for the meta-grammar chromosome than for the solution grammar chromosome. When we compare the chromosome lengths and expressed lengths for the mutation rate experiment no significant difference is observed.

The solution grammars seem to have quite a few paths for generating sentences. A path describes a sequence of productions which creates an output string. Combining the paths with the number of codons used gives the set of non-unique possible output strings. These are shown in Fig. 7.8. It can be seen that the possible output strings of the grammars are not infinite and are sometimes as low as 4.

In summary, the discussion has been focused on the grammar and on the structure of

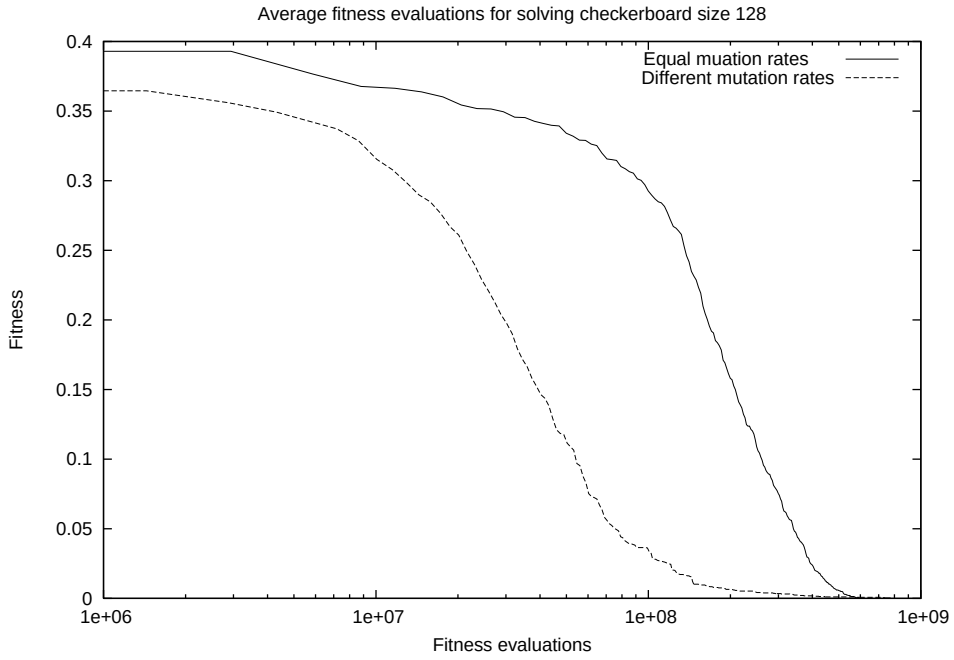
the solution grammar of individuals produced by the mGGA. This has shown that the output from the solution grammars can be quite restricted.

7.4 Summary

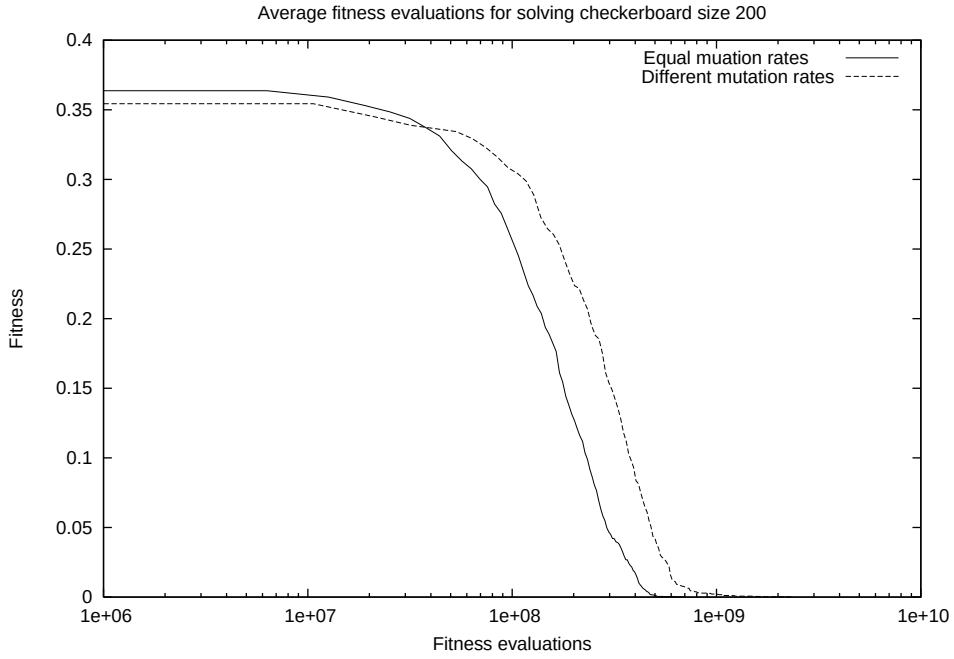
Adding a meta-grammar to Grammatical Evolution allows GE to evolve the grammar that it uses to specify the construction of a syntactically correct solution. The ability to evolve a grammar in the context of GE means that useful bias towards specific structures and solutions can be evolved during a run. This can lead to improved performance over the standard static grammar in terms of adaptation to a dynamic environment and improved scalability to larger problem instances. This approach of using building block structure rules in the grammar allows the evolution of modularity and reuse both on structural and symbol levels resulting in a compression of the representation of a solution. The meta-grammar increases not only the search space, but also a different grammatical bias. Therefore, there is a trade-off between the increase in search space and grammatical bias towards solutions with high fitness.

Two approaches were examined, the first adopts implicit sampling using different rates of mutation for the evolved solution grammar versus the solutions sampled from the evolved solution grammar. The second approach explicitly generates more than one sample from each solution grammar in a kind of local-search by randomly mutating the solution chromosome, which is used to construct sentences from the evolved solution grammar. For the problem instances examined neither approach was found to conclusively improve the performance of the meta-grammar approach to GE in terms of the number of fitness evaluations to find a solution. It is found that the majority of the evolutionary search is currently focused on the generation of the solution grammars to such an extent that the solutions are often hard-coded into them making the solution chromosome effectively redundant.

This leads us to Chapter 8 where there is one more question to address: when using meta-grammars, how does the mGGA solve the problems and did the bias in the meta-grammar affect its use of building block structures?



(a) Cb_{128_0}



(b) Cb_{200}

Fig. 7.4: The development of best fitness during the fitness evaluations. For different mutation rates and equal mutation rates. Log scale on x-axis and normalized y-axis.

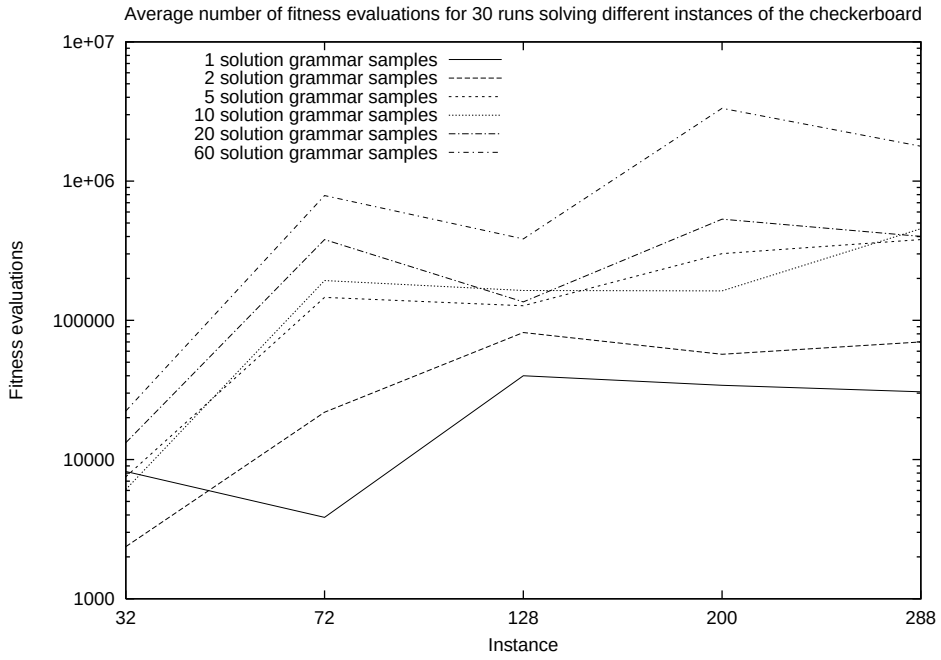
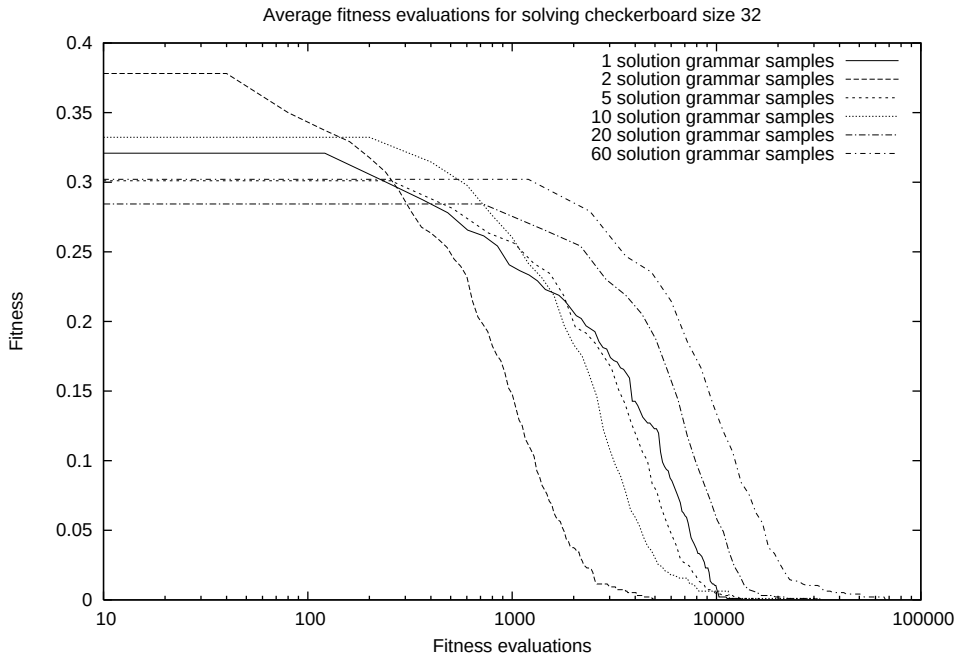
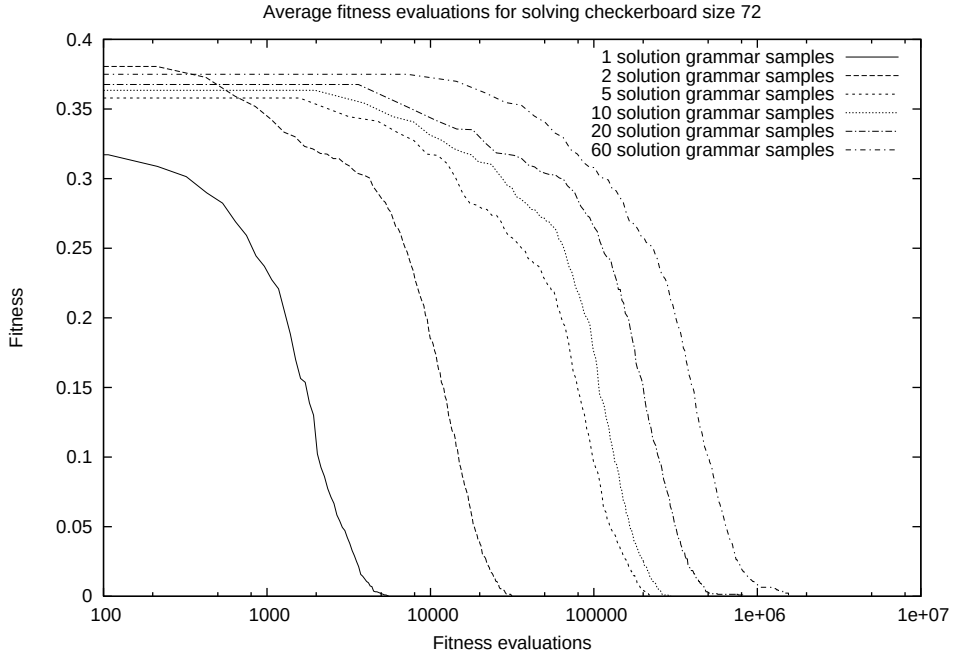


Fig. 7.5: On the x-axis are the problem instances, indicated by the total number of bits, and on the y-axis the number of fitness evaluations (log-scale).

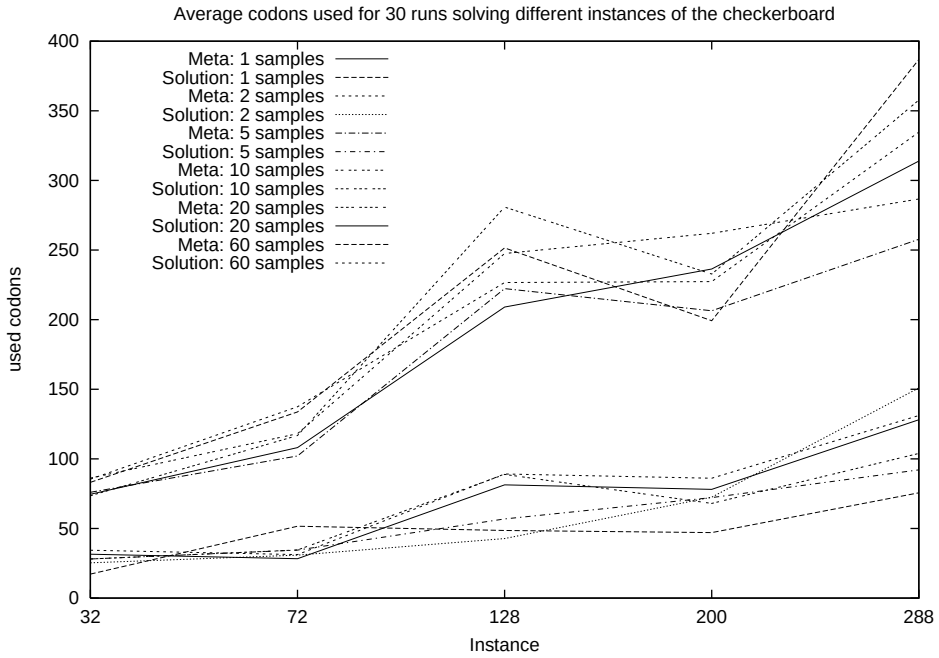


(a) Cb_{32}

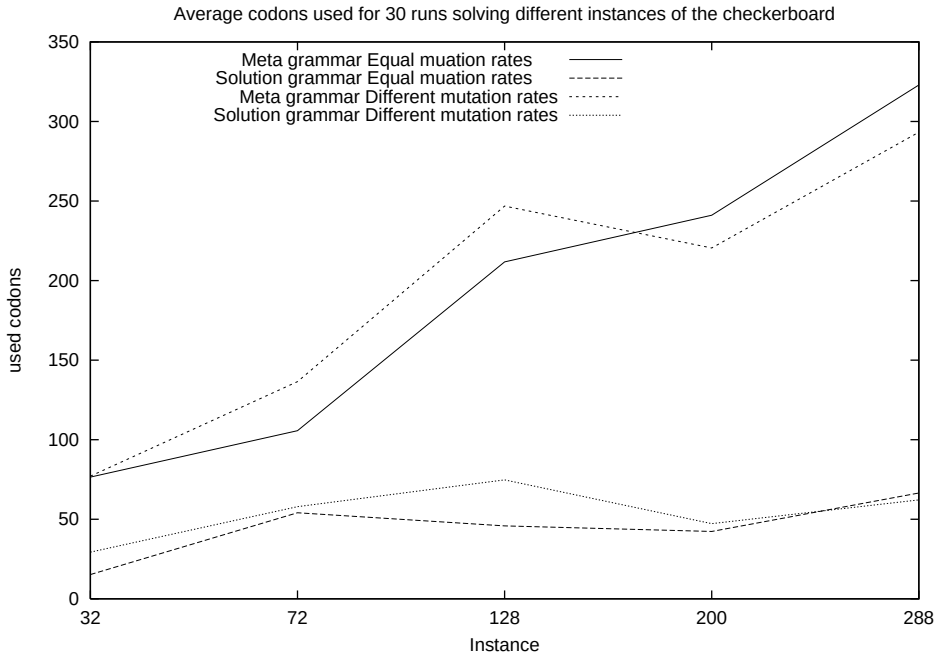


(b) Cb_{72}

Fig. 7.6: Note log scale on x-axis and normalized fitness on y-axis. Fig. 7.6(a) shows Cb_{32} and Fig. 7.6(b) shows Cb_{72}. The development of best fitness during the fitness evaluations for explicit sampling.

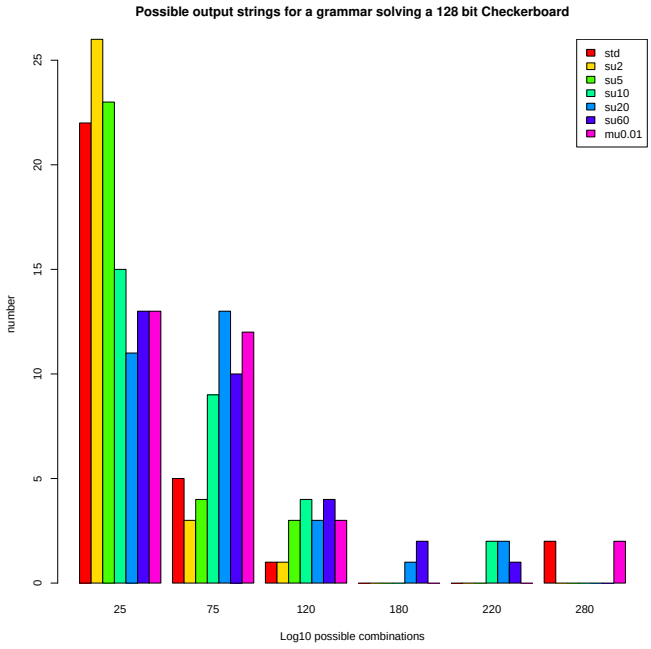


(a) Ave. length (n-samples)

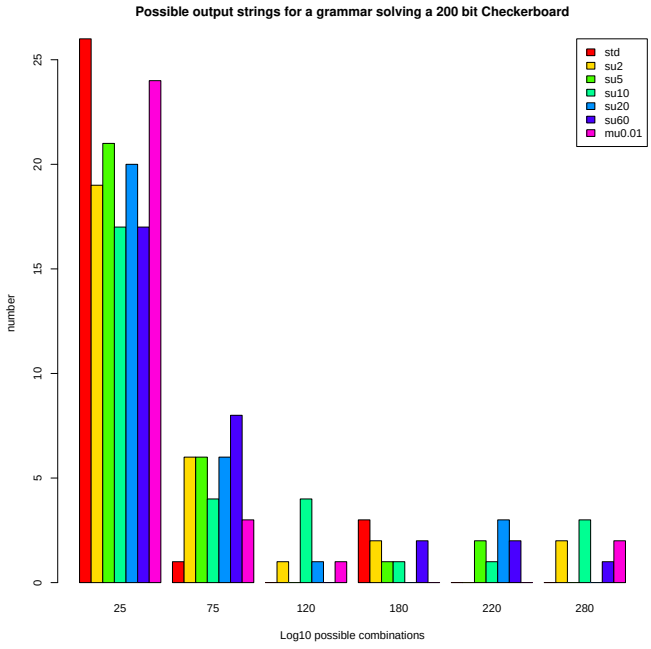


(b) Ave. length (Mutation)

Fig. 7.7: Average number of codons used at the end of each instance for n-samples in Fig. 7.1 and different mutation rates in Fig. 7.2.



(a) Cb_{128_0}



(b) Cb_{200}

Fig. 7.8: Histogram over the number of combinations of the possible paths in the solution grammars and codons used for Cb_{128_0} in Fig. 7.8(a) and Cb_{200} in Fig. 7.8(b). Samples with "Infinite" value are given the max value for each setting.

Chapter 8

Grammatical Bias and Use of Building Block Structures in the mGGA

Further investigation is done of the question of grammatical bias in the mGGA as well as to what extent the building block structures are used in the solutions generated by the mGGA, as investigated initially by Hemberg et al. [61]. There are studies of a series of grammars that incrementally change the bias for different parts in the grammar of the mGGA grammars adopted earlier, as well as how the building block structures that are being generated in the evolved grammars are used.

First, different grammars are studied in Section 8.1 and in Section 8.2 the experiments and results are presented. In Section 8.3 the discussion is concerned with how the use of building block structures in the individuals is investigated. Finally, a summary of the chapter appears in Section 8.4.

8.1 Grammar Design

There are many different configuration combinations when setting up a CFG. Here we explore how the different probabilities for choosing productions affect the performance of the mGGA. This is an analysis that will be of interest for a practitioner who attempts to design a grammar for experiments.

Whigham [160] investigates grammar learning by inducing grammar rules and probabilities for his GGP system, where two forms of bias are investigated, the declarative and the learned bias. The declarative bias of the grammars was modified to guide the search towards more specific solutions, leading to the argument that as the problem space grows, the language bias should be used to restrict the search space. Changing the grammar during the run is a learned bias in the problem space and operators are created that allow the encapsulation of new production rules. Each production has a fitness depending on how frequently they appear in the population. This makes the learned bias dependent on the usage of the rules and productions in the grammar.

Grammars used

A large number of grammars has been analyzed, with a series of incremental changes between each grammar. For clarity of presentation only the significant grammars are presented, and for each of the modified grammars only changes from the original grammar will be shown. The grammars differ as to how they bias towards building block structures. Both the building block structure biased grammar and the building block structure unbiased one refer to if there will be a uniform probability of choosing to use building block structures. The derivation tree generated is such that the rule `<g>` has a number of branches, which generates the number of repetitions, the number of different building block structures and content and the `<bit>` bias, e.g. see Fig. 5.3 on page 79. Each building block structure branch evolves the building block structure, where the `<bit>` is connected to the last branch, and the availability of building block structures is determined by the first branch. The grammars that were used in the experiments are:

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

Grammar 8.1: Part of Grammar 6.6 which biases the grammar towards the use of building block structures.

0. **Grammar 0 (Original)** To allow the creation of multiple building block structures of different sizes, the mGGAMBB is used as shown in Grammar 6.6 on page 100. When expanding `<bitstring>` there will be a bias towards using building block structures of size >1. In Grammar 8.1 it can be seen that each production in `<reps>` has uniform probability. Therefore, the probability for choosing `<bbk1>` is 1/3, which is lower than that for choosing a building block structure, 2/3. The probability for choosing `<bbk4>` is 1/3 and for `<bbk2>` it is 1/3.

1. **Grammar 1 (Equal 1)** Here the probability for `<bitstring>` to use Grammar 6.1 on page 96 or a building block structure of size >1 is equal, i.e. the probability for choosing a `<GA>...<GA>` structure is the same as for a building block structure, Grammar 8.2.

2. **Grammar 2 (Equal 2)** Shown in Grammar 8.3 is a grammar where the probability for `<bitstring>` to use a GEGA or building block structures of any size is equal, as in Grammar 1.

3. **Grammar 3 (GEGA)** This is a simple GE approach to GA that does not use a meta-grammar, see Grammar 6.1 on page 96. It is implemented in order to provide a benchmark for the other results. The grammar pre-specifies the number of bit positions in the solution, and the genome is used to select what each bit becomes.

```
<g> ::= "<bitstring> ::= <GA>...<GA>|" <reps>
              "<bb4> ::= " <bb4>
              "<bb2> ::= " <bb2>
              "<bit> ::=" <val>
               "<GA> ::= 1 | 0"
<reps> ::= <rept>
       |  "<GA>...<GA> |" <rept> "|" <reps>
<rept> ::= "<bb4><bb4><bb4><bb4><bb4><bb4><bb4><bb4>"
       | "<bb2>...<bb2>"
       | <reps>
```

Grammar 8.2: Grammar 1 has the same probability for `<bitstring>` to use a GEGA or a building block structure of size >1. *Italics* show differences with Grammar 0.

```
<g> ::= "<bitstring> ::= <GA>...<GA>|" <reps>
              "<bb4> ::= " <bb4>
              "<bb2> ::= " <bb2>
              "<bb1> ::= " <bb1>
              "<bit> ::=" <val>
              "<GA> ::= 1 | 0"
<reps> ::= <rept>
       | "<GA>...<GA> |" <rept> "|" <reps>
<rept> ::= "<bb4><bb4><bb4><bb4><bb4><bb4><bb4><bb4>"
       | "<bb2>...<bb2> "
       | "<bb1>...<bb1> "
       | <reps>
```

Grammar 8.3: Grammar 2 has the same probability for `<bitstring>` to use a GEGA or building block structures of any size. *Italics* show differences with Grammar 0.

4. **Grammar 4 (GEGABB)** This is a simple GE approach that does not use a meta-grammar, see Grammar 8.4 on the following page. The grammar allows selection of a building block structure which is expanded each time.

8.2 Experiments & Results

This section relates to the grammar designs on the Checkerboard problem and the dynamic noisy Checkerboard. We analyze two aspects of grammar design, the use of building block structure non-terminals and the impact of building block structure bias on performance.

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

Grammar 8.4: GE bit string grammar with building block structures <bbk4>, <bbk2>.

Hypothesis

The goal of these experiments is to determine if the bias in the meta-grammar is having a negative impact on the performance. The performance is measured as the average number of fitness evaluations required for 30 runs to solve an instance of the Checkerboard. ϕ_0 is the performance for the mGGA with Grammar 0 and ϕ_1, ϕ_2, ϕ_3, ϕ_4 is the performance for Grammar 1 and 2 and 3.

H_0**:** The performance ϕ_0 is the same as ϕ_1, ϕ_2, ϕ_3, ϕ_4

H_1**:** The performance ϕ_0 is not the same as ϕ_1, ϕ_2, ϕ_3, ϕ_4

α**:** The significance level of the test is 0.05.

Setup

For the experiments the population size was the one that solved the instance within 10% of where 30 runs are successful for a maximum of 800 iterations. For the Cb_{512} no population sizing was done, instead a fixed population size of 1000 was used. For the GEGA the parameters for the meta-chromosome were not used. The settings in Tab. 8.1 were the changes from Tab. 6.1.

A dynamic noisy version of the Checkerboard problem is also adopted. The Checkerboard was distorted with noise to make the problem harder [60]. The distortion was implemented by the possibility of each bit in the original board to flip with a probability

Tab. 8.1: Parameters for the GE algorithm

Parameter	Value
Checkerboard size	32, 128, 512
Period length	10, 100, 800
Initial chromosome size	90, 300, 1300
Population size	121, 288, 1000
Crossover probability	0.9 (Both chromosomes)
Mutation probability	0.01 (Both chromosomes)
Generations	800, 800, 800

of 0.05. Here it is taken a step further and the Checkerboard is distorted after a set period length of 10 or 100 iterations. Therefore, we have two dynamic variants of each problem size (Cb_{32} and Cb_{128_0}).

8.2.1 Checkerboard Results

Tab. 8.2 details the average generation for which a solution was found over the 100 runs (40 runs for Cb_{512}). It also details the results of a t-test on this data.

In the case of the simpler problem instance (Cb_{32}) both of the unbiased grammars, Grammar 1 and 2 significantly outperform the biased Grammar 0. This is not the case, however, for the larger Cb_{128_0} problem instance where statistically the results are the same. Examining the fitness plots in Fig. 8.1 we see that the unbiased grammars are solving the problem faster than the biased Grammar 0 for both problem instances. In Fig. 8.2 the fitness over generations for Cb_{512} is shown, it can be seen that the grammars with equal probability for using building block structures or not find the solution faster than Grammar 3 or Grammar 4. The variance for Grammar 0 is the highest, and it does not always find a solution. This confirms the statements that the right building block structures will allow you to find the solution faster, while with the wrong building block structures it can take longer, which can be seen by the increase in variance. In Tab. 8.3 the minimum, median and maximum values for the solution generations are shown for Cb_{512}.

An interesting result is that the simpler GEGA, as represented in Grammar 3, significantly outperforms all other grammars for both problem instances at the 95% confidence

Tab. 8.2: P-values of 2 sided t-test between the different grammars for the problem. Ave Gen. denotes at which generation(iteration) the problem was solved.

Cb_{32}

Grammar	Ave Gen.$\pm Std$	0	1	2	3	4
0	67.44 ± 22.64	x	**0.000**	**0.000**	**0.000**	**0.000**
1	46.90 ± 5.36		x	0.470	**0.000**	**0.000**
2	46.34 ± 5.58			x	**0.000**	**0.000**
3	41.42 ± 4.31				x	0.25
4	39.67 ± 4.79					x

Cb_{128_0}

Grammar	Ave Gen.$\pm Std$	0	1	2	3	4
0	136.96 ± 38.84	x	0.404	0.468	**0.031**	**0.016**
1	133.67 ± 6.17		x	0.648	**0.000**	**0.000**
2	134.09 ± 6.79			x	**0.000**	**0.000**
3	128.38 ± 6.97				x	0.300
4	126.39 ± 6.50					x

Cb_{512}

Grammar	Ave Gen.$\pm Std$	0	1	2	3	4
0	184.10 ± 265.113	x	0.74	0.80	**0.000**	**0.000**
1	170.45 ± 23.353		x	0.56	**0.000**	**0.000**
2	173.30 ± 20.920			x	**0.000**	**0.000**
3	579.60 ± 21.289				x	0.136
4	572.50 ± 20.597					x

level. For both problem instances it is also worth noting that the fitness standard deviation in the biased Grammar 0 is much higher than that in the unbiased grammars. This might be explained by the possibility for the population to converge on a building block structure with high fitness, or on one with initially lower fitness, which might slow down the search.

Phenotype Visualization

We examined the solutions evolved during the 30 runs at each generation and averaged the value at each position across the population. This result is visualized in Fig. 8.3 where a rapid convergence on the solution is found for all grammars.

Each column in the figure represents the *average* solution of that generation (i.e. the average value at each bit position of the solution) with the first column representing the first generation. It can be seen that the first generation is noisy (indicated by gray bit

Tab. 8.3: Min. solution generation, median solution generation and maximum solution generation for the different grammars for Cb_{512}

Grammar	Min Gen.	Median Gen.	Max Gen.
0	32	71	800
1	35	175	185
2	52	177.5	189
3	540	587	612
4	523	567.5	602

values at each locus) as would be expected from a random initialization process. As we move south through the figure (moving through each generation of a run) we see very rapid convergence on the ideal target solution of the Checkerboard (spread out across a line).

8.2.2 Dynamic Noisy Checkerboard Results

For the dynamic noisy Checkerboard instances similar trends are observed. The number of fitness evaluations for the runs are shown in Fig. 8.4(a) and 8.5(a). An alternative view of these results is to plot the area under the curve. In Fig. 8.5(a) and 8.5(b) we plot a simplification of the area under the curve by plotting a running total of the error at each period.

It can be seen from Fig. 8.4 and 8.5 that at this level of noise the grammars using building block structures react slower to changes in the fitness function than the Grammar 6.1. One reason for the changes in fitness is that the building block structures lose more fitness compared to Grammar 6.1. Rohlfshagen et al. [134] mention that it is possible to see the dynamics of changes in a OneMax problem as an additional mutation operator on the individual.

8.3 Building Block Structure Usage

Primarily, the experiments aim to assert if the evolved grammars and co-evolved solutions actually include and use building block structures of a size greater than one when solving a problem. To this end we compare the frequency of occurrence of the building block rules

in solutions with the frequency of occurrence of the same building block structures using a random search. To form an idea of the use of building block structures, one can look at the use of building block structures in the solution grammar of the individual that solved the problem and compare it to random samples. Fig. 8.6(a) and 8.6(b) show these results.

It can be seen in the presence of random search that there is a 50:50 split between solutions that adopt building block structures of size N versus building block structures of size 1 in Fig. 8.6(b). In the presence of an evolutionary algorithm however, the relative use of building block structures of size N is significantly greater than blocks of size 1. Size 1 building block structures are only used for 23% and 20% of the time for the Cb_{32} and Cb_{128_0} respectively. These results are encouraging and suggest that when a choice between adopting building block structures of size N versus no building block structures (i.e. size 1) is offered in the meta-grammar, building block structures will be exploited in solving the problem. The same trend can be observed for Grammar 2 for Cb_{32} and Cb_{128_0}.

8.4 Summary

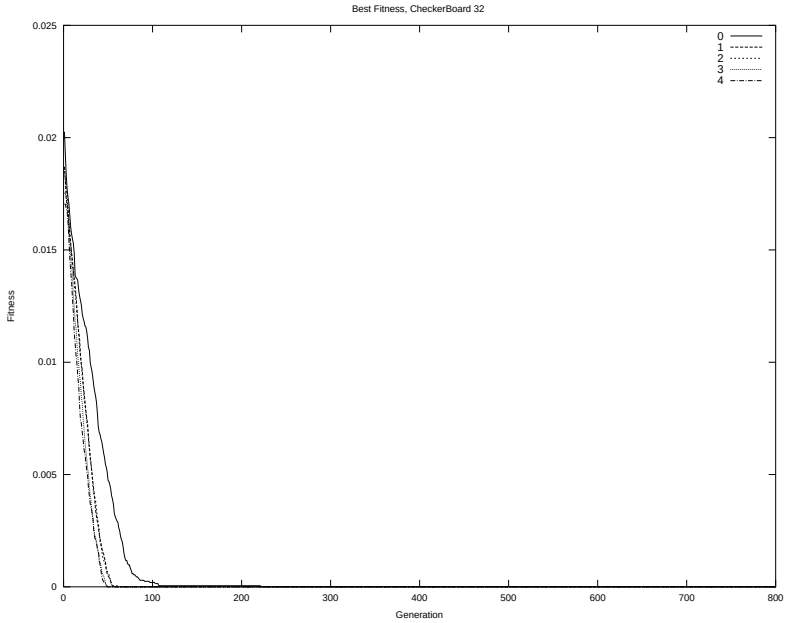
We set out to measure and understand two aspects of the meta Grammar Genetic Algorithm. Firstly, an experiment was undertaken to determine whether a bias in the grammar design used in earlier studies towards the use of building block structures impaired search efficiency. Secondly, we wished to determine whether the building block structures were in fact adopted by the population in solving the problem.

With respect to grammar design, it was found that this can be an important factor in the search efficiency of the meta-grammar approach of the problems analyzed. A grammar which had equal probability of whether or not to use a building block structure was found to outperform the biased equivalent. An analysis of the adoption of building block structures by the evolutionary search found that these modular structures were used successfully by the population to solve the problem. A recommendation arising from this experiment is towards the adoption of a meta-grammar that allows the use of both a classic GA bitstring

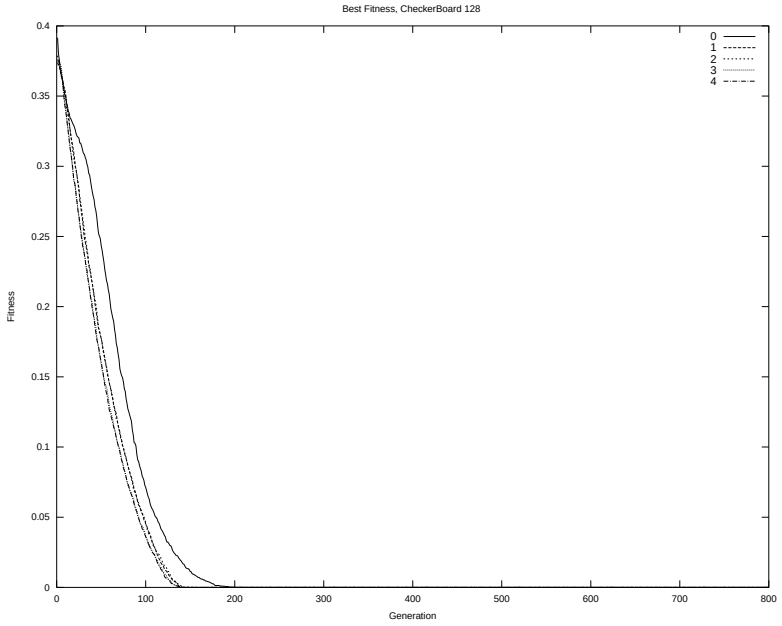
representation in conjunction with the modular building block structures.

This chapter concludes Part II which has explored grammars in the GE framework in Chapter 4 by investigating the mapping order of these grammars as well as meta-grammars, where in particular modularity in Chapter 5, scalability in Chapter 6, operators in Chapter 7 and grammar design in Chapter 8 were studied. The meta-grammars have a larger search space in comparison to the standard GE grammars, as well as a different representation with two chromosomes. The ability of the meta-grammars to define different building block structures can balance the increase in search space size with an increase in performance, given the bias from the meta-grammar. The meta-grammar representation can also benefit from different mutation rates on the chromosomes, thereby allowing a slower rate of evolution on the meta-chromosome. The use of building block structures increases the variance of the performance, since beneficial building block structures will discover the solution fast, and misplaced building block structures can be slow.

In Part III our empirical exploration is further analyzed in an attempt to generalize and formulate some theory regarding grammars in GE. Especially how an individual solution changes when the codons are changed, how the structures are preserved in the population and how to measure grammars.



(a) Cb_{32}



(b) Cb_{128_0}

Fig. 8.1: On the x-axis is the number of fitness evaluations. On the y-axis is the normalized fitness for the different grammar versions. Fig. 8.1(a) shows Cb_{32} and Fig. 8.1(b) shows Cb_{128_0}

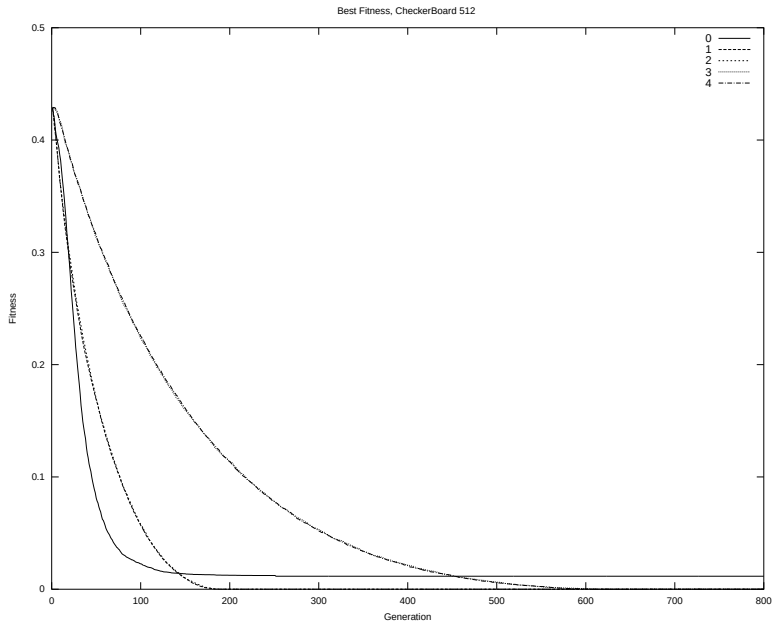
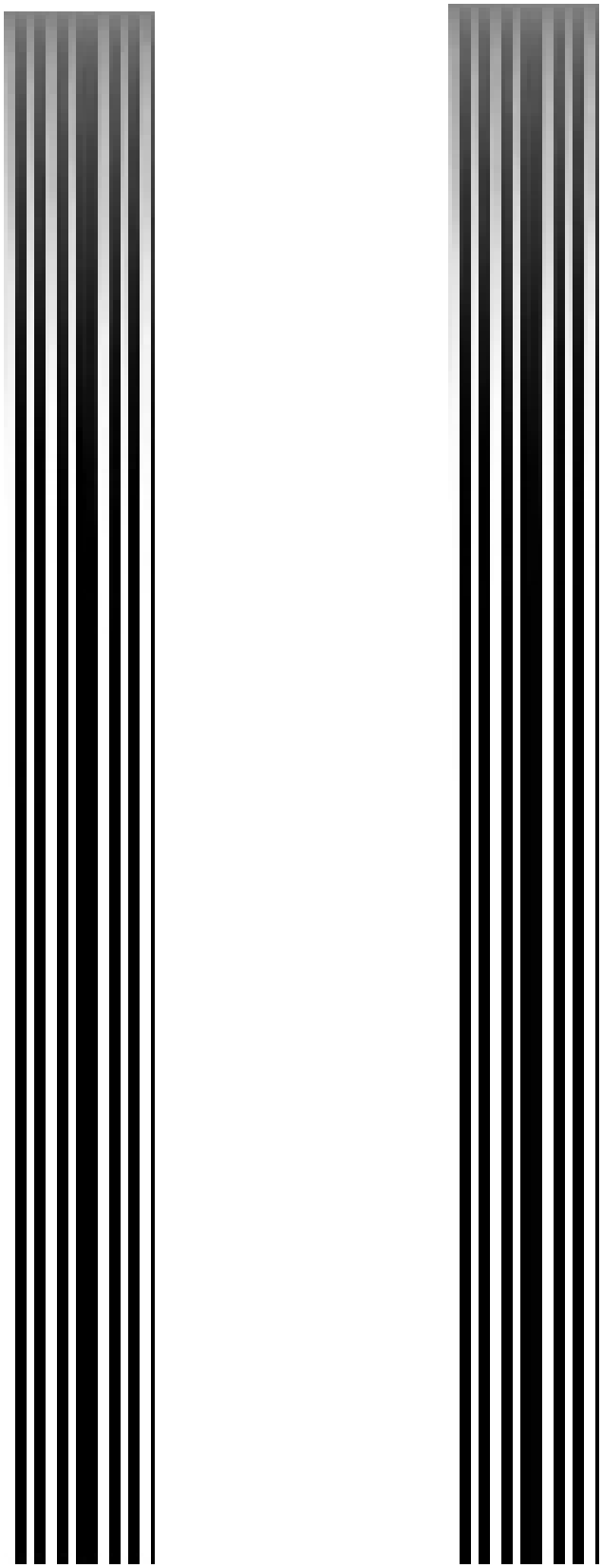
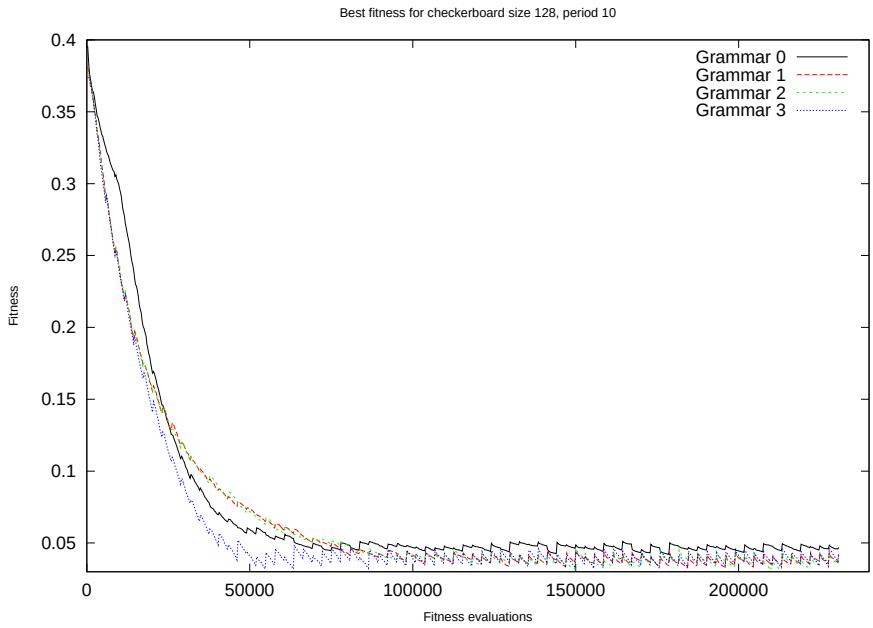


Fig. 8.2: On the x-axis is the number of generations. On the y-axis is the normalized fitness for the different grammar versions for Cb_{512}

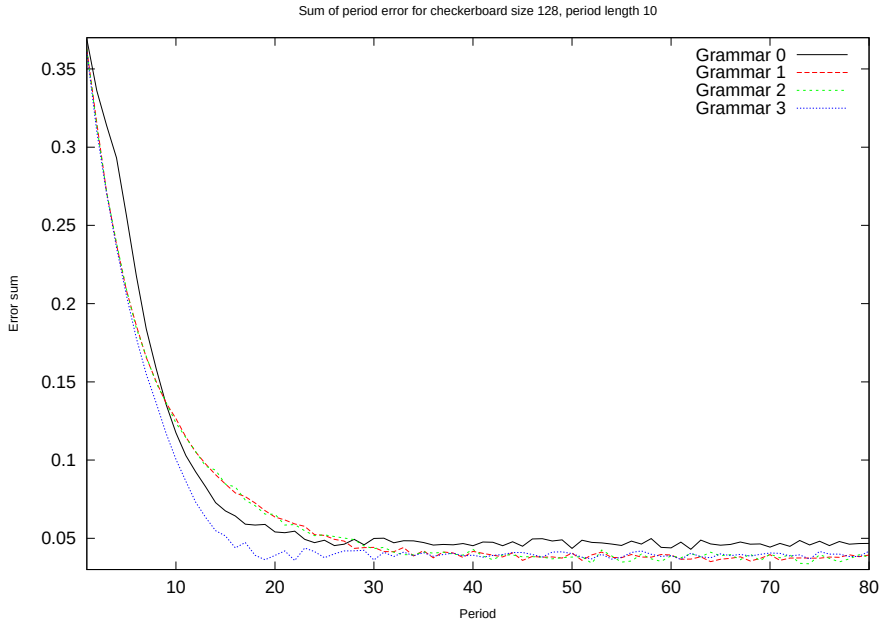


(a) Gram. 0 (b) Gram. 2

Fig. 8.3: The appearance of solutions for a sample of 100 runs of Cb_{32}, Fig. 8.3(a) is Grammar 0 and Fig. 8.3(b) is Grammar 2. The x-axis is the average preformance at each locus in the population, the y-axis is the generation (300 in total). Convergence of the bit values occurs approximately within the first 50 generations.

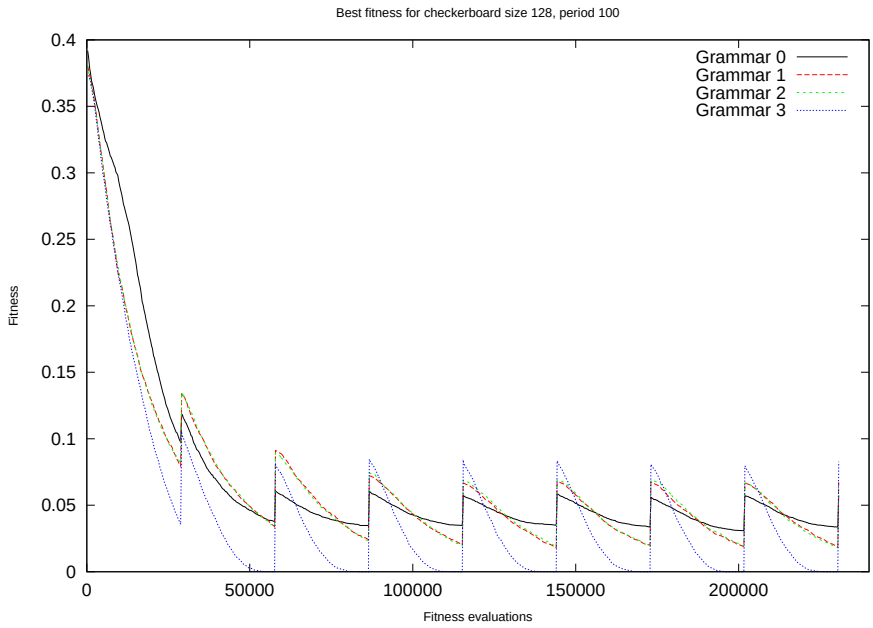


(a) Cb_{128_0}, period 10

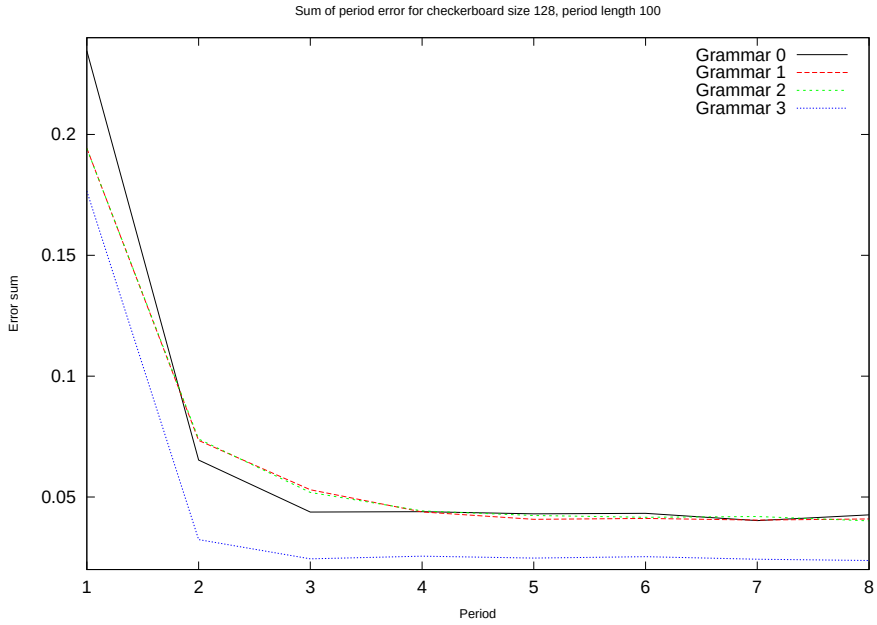


(b) Cb_{128_0}, sum-of-errors, period 10

Fig. 8.4: In Fig. 8.4(a) the x-axis is the fitness evaluation and the y-axis is the normalized fitness and in Fig. 8.4(b) the x-axis is the period and the y-axis the sum of errors for each period, for the different grammar versions

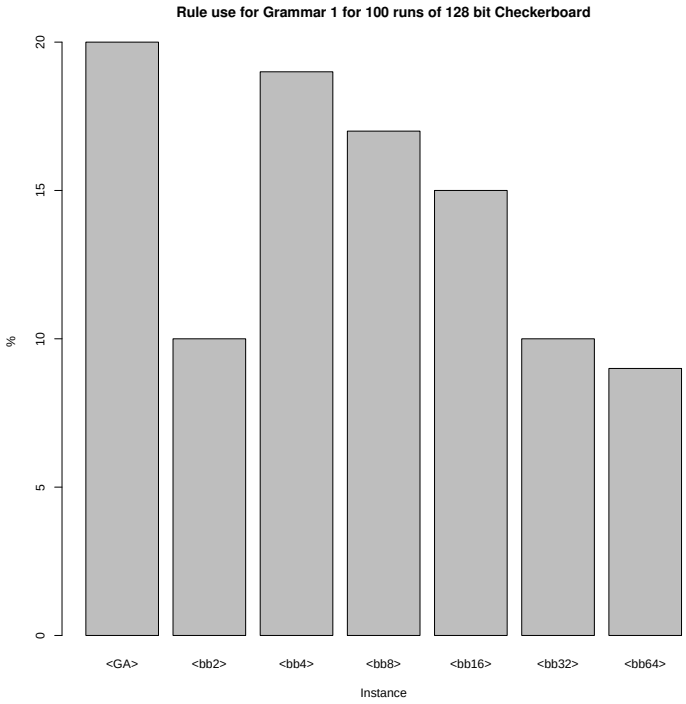


(a) Cb_{128_0}, period 100

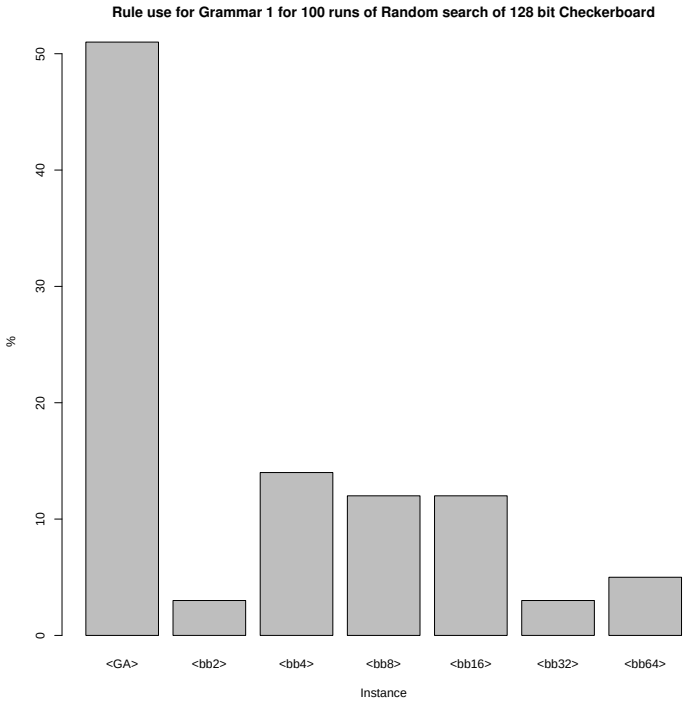


(b) Cb_{128_0}, sum-of-errors, period 100

Fig. 8.5: In Fig. 8.5(a) the x-axis is the fitness evaluation and the y-axis is the normalized fitness and in Fig. 8.5(b) the x-axis is the period and the y-axis the sum of errors for each period for the different grammar versions



(a) Cb_{128_0}, Grammar 1, mGGA



(b) Cb_{128_0}, Grammar 1, Random search

Fig. 8.6: Use of building block structures in solution for mGGA Cb_{128_0} for Grammar 1 in Fig. 8.6(a) and using Grammar 1 and random search in Fig. 8.6(b).

Part III

Theory - Formalizing the Exploration of Grammars in Grammatical Evolution

After empirically exploring grammars in GE in Part II, where the experiments of the mapping order of a grammar and the performance of meta-grammars as well as alterations in the search were studied, we now have data and will proceed with a formal description and theory regarding GE in Part III. In Chapter 9 we formalize and describe the mapping process in order to break down the grammatical and search bias which is produced by the mapping. This will allow further analysis of the search, as well as simplifying the analysis. Moreover, in Chapter 10 we investigate changes in input and the effect on output and analyze the neighboring solutions and the effect of changes (and bias in representation), of both single and multiple changes. Defining different types of changes allows classification of the effects that input changes (operators) have. The changes are a part of the identification of what the neighborhood looks like. Furthermore, a schema theorem for investigating preservation of material during variation, an attempt to identify the population effects will be presented here. Finally, in Chapter 11 we investigate grammar measurements, and how to compare grammars.

Chapter 9

Formal Description of GE and the mGGA

The aim of this chapter is to provide complementary insights into the GE and mGGA algorithms. GE and in the process mGGA are approached with a more formal description with inspiration from previous works on grammars and mapping in EC [151, 159, 163]. By investigating the grammatical representation and the operators separately the essence of GE and the mGGA can be more clearly understood. Also implications of declarative bias coming from the use of a generative grammar representation and the constraints of the algorithm are shown from a different angle.

Initially the description will be of GE and will later be extended to the mGGA. Moreover, the effects of grammar mapping and disruptions in the input will be explored, showing two distinct change effects from alterations of the input, and subgroups of these effects. The effects are classified depending on changes in the subtree size.

As seen in Chapter 4 different grammars which impose different mapping orders have different search behaviors. When understanding and expanding GE, which uses mapping, it can be useful to see where explicit bias and different impacts of change in one space will occur and how these relate to the space they are mapped to and also which operators should be used. Thus we can understand how changes in input are translated to alterations

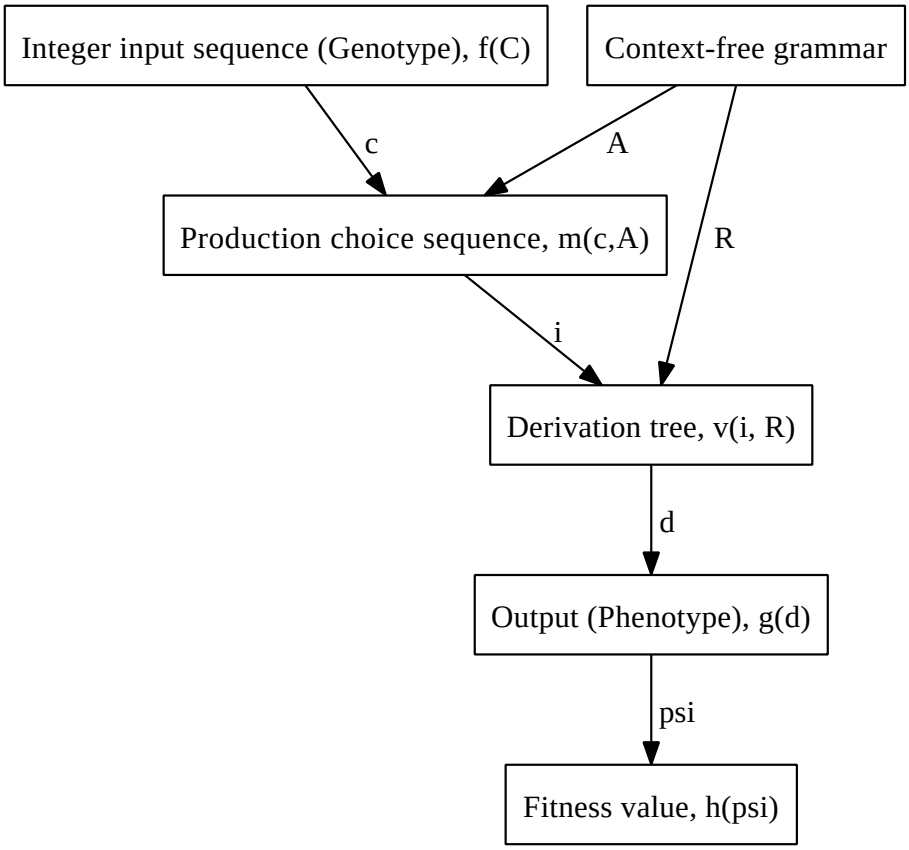


Fig. 9.1: Grammatical Evolution spaces. In GE the grammar maps the input (genotype) to the output (phenotype) which is evaluated. During this process several mappings between different spaces are made. The arrow labels indicate the mapping from one space to another.

in output. This section lists some of the spaces in the GE algorithm; see Fig. 3.5 on page 48 for a visualization of the GE algorithm and the mapping in Fig. 9.1 (which will be explained more thoroughly in Section 9.1) .

In GE there is redundancy in more than one of the mappings, i.e. a many-to-one mapping between input and output. The grammar has an impact on both the derivation of the output (phenotype) and the final phenotype, e.g. on non-terminals and rules, and on the language that the terminals can generate, revisiting the meta-grammar experiments. The GE evaluation, consisting of genotype-phenotype (input-output) mapping and fitness assignment, can be described with the following spaces for the GE individual [1].

[1]Canonical GE has a binary chromosome that will be transcribed to integers.

1. Integer input sequence (genotype or more explicitly the chromosome)
2. Production choice sequence. Mapped by the current rule in the grammar
3. Derivation tree, the production choice sequence represented as a derivation tree. Mapped by the rules in the grammar
4. Output sentence (phenotype), the leaves of the rewritten derivation tree
5. Fitness, the evaluated phenotype

The structure of this section consists of a formal description of the components of GE in Section 9.1 Section 9.2 describes the mGGA and Section 9.3 discusses the implications. In Section 9.4 the findings of the chapter are concluded and summarized.

9.1 The GE Components

We start by describing the components of GE mapping more formally than in Chapter 3 on page 33, in order to illustrate the mapping bias mediated by the CFG from input to output. It is a different phrasing of Ex. 7 on page 41. The description is similar.

The input sequence (genotype) consists of chromosomes which are comprised of codons. In GE a chromosome is a sequence of integers[2] [3]

Definition 12 (Individual in GE) An individual in GE is a single chromosome C, a sequence of codons. Each codon uses m bits to encode an integer: this gives codon values in the range $[0, 2^m - 1]$:

$$C = \langle c_0, \ldots, c_n \rangle,\ c_i \in \mathbb{Z}_{2^m},\ 0 \leq i \leq n,\ m, n \in \mathbb{N} \qquad \square$$

[2]Canonical GE has a binary chromosome that will be transcribed to integers, $f : \mathbb{Z}_2 \to \mathbb{Z}_{2^m},\ f(C_2) = C$ where m is the codon size and C_2 the binary representation of the chromosome. Here we simplify and skip the transcription step, binary-to-integer, and use a sequence of integers instead.

[3]$\mathbb{N}$ refers to the natural numbers, $\mathbb{Z}$ denotes integers, and $\mathbb{Z}_n$ integers modulo n.

Example 9 (GE individual) A GE individual of length four, $|C| = 4$, might have a chromosome of $C = \langle 42, 666, 13, 49\rangle$ which is represented by the primitive int type in Java (32 bits) $-2^{31} \leq c_i \leq 2^{31} - 1, m = 32$. □

Now that we know what a GE individual is we can study the different mappings that will occur.

Chromosome to Production Choice Sequence – Integer Sequence to Integer Sequence

In the mapping, the leftmost non-terminal is expanded and the value of the current codon decides which production to choose for the expansion of the rule, Section 3.1.3 on page 38. A Context-Free Grammar is denoted by G (Def. 1 on page 16) and a Probabilistic Context-Free Grammar is the tuple $\langle G, P\rangle$ (Def. 5 on page 19). We denote the space of Probabilistic Context-Free Grammars as $\langle G, P\rangle \in \mathcal{G}$. The first step of the mapping in the GE system can be described as a chromosome C and a grammar $\langle G, P\rangle$ generating a production choice sequence of production choices $I = \langle i_0, \ldots, i_n\rangle, n \in \mathbb{N}$. Alternatively, there is a mapping between two integer spaces, lists of integers to lists of integers with max value being the maximum number of production choices of all the rules in the grammar.

$$f : \langle \mathbb{Z}_{2^m}, \ldots, \mathbb{Z}_{2^m}\rangle \times \mathcal{G} \rightarrow \langle \mathbb{Z}_{r_{max}}\rangle, f(\langle G, P\rangle, C) = I \tag{9.1}$$

For the integer input sequence $r_{max} = |R|$ we use unique identifiers for each production choice up to the number of productions in the grammar[4] and I can also be written as a derivation tree D.

Example 10 (Input of production choices) A GE individual with the codon input sequence $C = \langle 42, 666, 13, 49\rangle$ and grammar from Grammar 2.1 $\langle G, P\rangle$ has the input

[4]In canonical GE the maximum number of production choices is $r_{max} = max_{r_{i*} \in R}(|r_{i*}|)$. If r_{max} uses unique identifiers the analysis can be facilitated. The context and analysis should make it obvious when the integer inputs are unique or ambiguous.

Tab. 9.1: Codon change and production choice type change. These are the impacts from a redundant deterministic mapping.

Type	**Codon**(c)	**Production choice**(i)
Unchanged	$c = c'$	$i = i'$
Redundant	$c \neq c'$	$i = i'$
Change	$c \neq c'$	$i \neq i'$

sequence $\langle 0, 0, 1, 1 \rangle$ and $r_{max} = 2$ with $|I| = 4$ or $\langle 0, 1, 2, 2 \rangle$ and $r_{max} = 4$. Note that deterministic choices do not use codons. □

The types of changes that are possible from the mapping from codon sequence to input sequence, $f(C) = I$ are shown in Tab. 9.1. The cases $c = c'$ and $i \neq i'$ are not possible when using a deterministic mapping. (' denotes the next time step, not only change)

To ensure that a correct production is chosen the modulo of the codon value is used to decide the corresponding production, as mentioned in Chapter 3 on page 33. The integer value of the codon is mapped to an integer value in the range of the number of production choices for the rule, $[0, |A_k|]$. In other words, the mapping decides which production choice to use, given the derivation step $k, 0 \leq k$ for index of the current rule (non-terminal to expand) and j for the index of the codons c_j. To select via the many-to-one mapping from codon integer value and non-terminal to an integer representing the production choice

$$m : \mathbb{Z}_{2^m} \times N \rightarrow \mathbb{Z}_{r_{max}}, m(c, A) = i \tag{9.2}$$

denoting $i_{k_a} = m(c_j, A_k)$ with

$$m(c_j, A_k) = c_j \bmod |A_k| \qquad\qquad A_k \in N, c_j \in C \tag{9.3}$$

where $|A_k|$ is the number of productions corresponding to non-terminal A_k as in Eq. (2.1 on page 20). To get a unique identification for i_k then

$$u : \mathbb{Z}_{|A_k|} \times \mathcal{G} \times \mathcal{G} \rightarrow \mathbb{Z}_{r_{max}}, u(i_{k_a}, G, A_k) = i_k \tag{9.4}$$

The point of the following equation is to clarify the derivation order in which the phenotype is generated by the grammar. In canonical GE the derivation sequence is read from left-to-right. A_k is the leftmost non-terminal in the derivation δ_k. The sequence of non-terminals in derivation δ_k is denoted δ_k^N and the index i is written $\delta_{k,i}^N$

$$\begin{aligned} A_{k+1} &= \delta_{k,0}^N \\ \delta_k^N &= \{x : x \in \delta_k,\ x \in N\} \end{aligned} \tag{9.5}$$

and the initial derivation starts with the start symbol, $\delta_0 = S \Rightarrow A_0 = S$.

The purpose of this paragraph is to show the discrepancy in codons used and integer input length. The derivation step k indicates the current index in I as well as the number of times the genotype C is wrapped and reread from the beginning, $0 \leq k \leq |C|w,\ w \in \mathbb{N}$. The wrapping is expressed as the modulo of the genotype length taken from the derivation step counter, k and gives the chromosome index j,

$$j = k \bmod |C| \tag{9.6}$$

Moreover, note that a new input from codon c_j is only read when the current rule, A_k, is non-deterministic, i.e.

$$j := \begin{cases} j, & \text{if } |A_k| = 1 \\ j+1, & \text{if } |A_k| > 1 \end{cases} \tag{9.7}$$

This gives the relation between the used codon sequence and the input sequence, i.e. the index for codon c_j and integer input i_k has $j \leq k$, as shown in Eq. (9.6) and Eq. (9.7).

Example 11 (Mapping the codon to a production choice) A derivation using the grammar from Ex. 1 on page 17 and input codon sequence $C = \langle 42, 666, 13, 49, \dots \rangle$. The derivation from the beginning, when $j = 0, k = 0$ and $\delta_0 = A_0 =$ <bitstring> from

Eq. (9.3), $m(c_j, A_k)$ is

$$\begin{aligned} i_0 &= m(42, \texttt{<bitstring>}) \\ &= 42 \bmod |\texttt{<bitstring>}| \\ &= 42 \bmod 1 \\ &= 0 \end{aligned}$$

This gives the derivation string for step $k = 1$, from Eq. (9.5)

$$\begin{aligned} \delta_1 &= \underline{\texttt{<bbk4>}}\texttt{<bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4>} \\ \delta_1^N &= \{\texttt{<bbk4>,<bbk4>,<bbk4>,<bbk4>,<bbk4>,<bbk4>,<bbk4>,<bbk4>}\} \end{aligned}$$

and the next non-terminal to expand is $A_1 = \texttt{<bbk4>}$. Furthermore, since there was only a single production to choose from, no codon was used and the input codon sequence index was not increased since Eq. (9.7) gives, $|A_0| = 1 \Rightarrow j = 0$. □

In order to avoid too much bias from the production choice mapping in Eq. (9.3) a large enough difference between the max codon value and the number of production rules is required, $2^m - r_{max} >> 0$. With this assumption the modulo operation in the Eq. (9.3) makes all the probabilities uniform for the current rule $p_{ij} = 1/|A_k|, p_{ij} \in P$, with the probability to be selected $1/|A_j|$, see Ex. 1 on page 17. Otherwise there is a bias when the modulo rule is applied to the codon which makes some i_j more probable than others [74, 163]. Moreover, GE can be seen as a PCFG (Def. 5 on page 19), where all the probabilities are uniform, i.e. the probabilities are determined by the number of production choices for each non-terminal, $p_{ij} = 1/|r_{i*}|$. A bias towards productions r_{ij} can be achieved by multiple identical production choices in the solution grammar rules.

The following analysis describes very generally what types of changes occur when the chromosome changes, and how the grammar impacts them. The mapping introduced the grammar, by Eq. (9.3) increases the number of parameters from the basic case described

Tab. 9.2: Codon change, current rule and production choice type change. Deterministic neutral mapping and dependency impact. The input i denotes production choices, c the codon value and A the current rule. Change can occur, depending on the grammar, in between zero and three of the five cases

Type	**Production choice**(i)	**Codon**(c)	**Parent**(A)
Change/Redundant	$i \neq i'$	$c \neq c'$	$A \neq A'$
Change/Redundant	$i \neq i'$	$c = c'$	$A \neq A'$
Change/Redundant	$i \neq i'$	$c \neq c'$	$A = A'$
Redundant	$i = i'$	$c \neq c'$	$A = A'$
No Change	$i = i'$	$c = c'$	$A = A'$

in Tab. 9.1. The mapping from codon to input is first extended with a dependency on the current rule i.e the parent, which comes from the grammar. If the current rule changes we call this a change of derivation context. Basic types of change to the production choice are shown in Tab. 9.2. The dependency introduced by the current rule sets the input in a context where the position of the integer value is important, which makes the table itself not directly applicable to GE. Change/Redundant represents the types that occur due to the position dependency. Furthermore, in BNF form the grammar can have duplicated rules. This leads to the possibility of change occurring, depending on the grammar, in three of the five cases. The number of cases where change can be directly caused by a codon change is two. An example of a grammar that would have a maximum probability of change would be one without duplicated rules or production choices, since then there would be no redundant changes.

Genotype to Phenotype – Integer Sequence to Word

For GE, one defining aspect is the redundancy in the mapping (many-to-one). With the grammar mapping in GE it is not guaranteed that the language the grammar produces will be able to produce all possible solutions. The mapping $\zeta : \langle \mathbb{Z}_{2^m}, \ldots, \mathbb{Z}_{2^m} \rangle \rightarrow \Psi$ from the chromosome in genotype space to the sentence in the solution space generates a solution (sentence) in the language generated by the grammar $\psi \in L(G) \subseteq \Sigma^*$. This is a property of the grammar and allows declarative bias to parts of the solution space. Ideally the grammar

Tab. 9.3: Codon change, production choice, current rule and next rule type change. The table shows the deterministic neutral mapping and dependency impact. The interval for change for the eight cases is between three cases and six cases. Delayed/Redundant implies that the change might be delayed since the expanded non-terminal is the same.

Type	**Rule**(A_{k+1})	**Codon**(c_j)	**Parent**(A_k)	**Production Choice**(i)
Change	$A_{k+1} \neq A'_{k+1}$	$c_j \neq c'_j$	$A_k \neq A'_k$	$i \neq i'$
Delayed/Redundant	$A_{k+1} = A'_{k+1}$	$c_j \neq c'_j$	$A_k \neq A'_k$	$i \neq i'$
Change	$A_{k+1} \neq A'_{k+1}$	$c_j = c'_j$	$A_k \neq A'_k$	$i \neq i'$
Delayed/Redundant	$A_{k+1} = A'_{k+1}$	$c_j = c'_j$	$A_k \neq A'_k$	$i \neq i'$
Change	$A_{k+1} \neq A'_{k+1}$	$c_j \neq c'_j$	$A_k = A'_k$	$i \neq i'$
Delayed/Redundant	$A_{k+1} = A'_{k+1}$	$c_j \neq c'_j$	$A_k = A'_k$	$i \neq i'$
Redundant	$A_{k+1} = A'_{k+1}$	$c_j \neq c'_j$	$A_k = A'_k$	$i = i'$
No Change	$A_{k+1} = A'_{k+1}$	$c_j = c'_j$	$A_k = A'_k$	$i = i'$

biases to a subset of the solution space, $V^* \subseteq \Psi$ where an optimal solution exists. If there is no knowledge of which solutions should be constrained the grammar can cover the entire solution space, $V^* = \Psi$. Of course, it should be avoided to constraint the grammar to a region without optimal solutions. Finally, the mapping ζ of GE $\zeta : \langle \mathbb{Z}_{2^m}, \ldots, \mathbb{Z}_{2^m} \rangle \rightarrow \Psi$ is many-to-one (depending on Ψ).

The space of derivation trees is denoted $\mathcal{D}$. The input sequence can be written as a derivation tree D

$$v : \langle \mathbb{Z}_r \rangle \times \mathcal{G} \rightarrow \mathcal{D}, v(I, \langle G, P \rangle) = D \tag{9.8}$$

where v is one-to-many if regarded out of the context dependency. In graph theory a tree D is a directed graph without cycles and is the ordered triple of a set of vertexes (or nodes), a set of edges and a mapping from the set of edges to ordered pairs of vertexes $D = \langle \gamma, \epsilon, \upsilon \rangle$. The transformation from input to production rule, $v(I, \langle G, P \rangle) = D$. When studying how a grammar reacts to changes in Tab. 9.3 the change in a codon and how it is related to the production choice are shown. The interval for change for the eight cases is between three and six cases. Delayed/Redundant is indirect since it implies that the change might be delayed since the expanded non-terminal is the same. When reading from left to right

```
<A> ::= <A><B>
      | <B>
<B> ::= c | d
```

Grammar 9.1: Left-recursive grammar which can have an immediate change effect

```
<A> ::= <B><A>
      | <B>
<B> ::= c | d
```

Grammar 9.2: Right-recursive grammar which can have an delayed change effect

a delayed change occurs in right recursive grammars $A \rightarrow A\alpha$. Delayed implies that the change in output (phenotype) might be delayed since the expanded non-terminal is the same.

Example 12 (Delayed changes of production) An example of a left-recursive grammar which can have an immediate change if the production choice changes is shown in Grammar 9.1. The sequence $\langle 0, 1, 0, 0\rangle$ gives `cc` and the changed sequence $\langle 1, 1, 0, 0\rangle$ gives `d`. A right- recursive grammar which can only give a delayed change is shown in Grammar 9.2. The sequence $\langle 0, 0, 1, 0\rangle$ gives `cc` and the changed sequence $\langle 1, 0, 1, 0\rangle$ gives `c`. Grammar 9.2 maintains the first terminal, in contrast to Grammar 9.1. □

Preservation from a delay or shift should occur when the parent is not the same, but the rule A_{k+1} is, i.e. when there has been a previous change but the output at this point stays the same.

The derivation tree D generates a derivation δ_T via a many-to-one mapping g (depending on the grammar g it can be one-to-one), i.e. the derivation tree is collapsed and the leaves read from left to right give the phenotype, or the derivation at the last step T, δ_T. The space of grammar symbols is $\mathcal{V}^*$

$$g : \mathcal{D} \rightarrow \mathcal{V}^*, g(D) = \delta_T \tag{9.9}$$

The mapping from g has the same properties as in Tab. 9.1. In the step from the final

derivation to a solution (phenotype) $k : \mathcal{V}^* \rightarrow \Psi, k(\delta_T) = \psi$

$$\psi = \begin{cases} \delta_T, & \text{if } \delta_T \in \Sigma^* \\ \text{Undefined}, & \text{otherwise} \end{cases} \tag{9.10}$$

Note that we assume that the sentences that are created by the grammar are defined in the solution space $\Sigma^* \subseteq \Psi$.

Output to Fitness Value

The phenotype, ψ is assigned a real value by evaluating it with a fitness function, $h : \Psi \rightarrow \mathbb{R}, h(\psi) = \phi$. It is difficult to generalize the properties of a fitness function; the mapping is often many-to-one, but ideally the fitness function should map from the phenotype space to the real values one-to-one and onto.

$$h(\psi) = \begin{cases} \phi, & \text{if } \psi \in \Psi \\ \phi_{min}, & \text{if } \Psi \text{ is Undefined} \end{cases} \tag{9.11}$$

This gives individuals that were not mapped completely minimum fitness, ϕ_{min}. The entire process in this section is shown in Fig. 9.1 on page 146.

Combining Eqs. (9.1), (9.9) and (9.11) for each input codon sequence gives the following expression for mapping and evaluating GE, see Fig. 9.1 on page 146. Given a grammar, and probabilities of production choices in the grammar and codons, a fitness value *phi* is calculated as

$$\phi = h(k(g(v(m(f(C), \langle G, P\rangle), \langle G, P\rangle)))) \tag{9.12}$$

To summarize, this section has introduced a formal description for the components of the GE mapping.

9.1.1 Representation Spaces in GE

Here we highlight the many different representation spaces that are passed through during GE mapping, as seen in Section 9.1. When understanding an algorithm that uses mapping it is useful to know where there will be different implicit and explicit bias as well as to which space a certain evolutionary operator can be successfully applied, in order to fully understand how changes in input will affect output. This section lists some of the spaces in the GE algorithm, see Fig. 9.1 on page 146.

The GE evaluation, consisting of input-output mapping and fitness assignment, can be described with the following spaces for the individual, showing that there is a redundant (many-to-one) mapping between more than one pair of spaces. Changed values are denoted by $'$, e.g. after a change to chromosome C the new chromosome is C'. $\Delta(x, x') = y,\ y \in \mathbb{R}$ denotes a function measuring a difference. The spaces in GE where changes are examined are:

1. The chromosome which has chromosome changes, $0 < \Delta_C(C, C')$ (here, Δ_C can be the Hamming distance).

2. The production choice sequence of integers has changes, $0 \leq \Delta_I(I, I')$

3. The derivation tree has changes, which are grammar dependent, $0 \leq \Delta_D(D, D')$ i.e. there are grammars that generate identical derivation trees (Δ_D could be the tree-edit distance).

4. The phenotype changes are the changes in the derivation tree leaves $0 \leq \Delta_\Psi(\Psi, \Psi'))$. In other words, there are grammars where different derivations give the same phenotype.

5. The fitness changes, which are $\Delta_\phi(\phi, \phi') \in \mathbb{R}$, different phenotypes can have the same fitness.

Note that for example mappings can be added between step 4 and 5, e.g. semantics, but these are not investigated in this thesis. In GE, there is redundancy in more than one

of the mappings; one point clearly shown is the impact of the grammar both on the input and the derivation, i.e. N and R, and on possible phenotypes $\psi \in \Sigma^*$.

9.1.2 Full Description of GE

The aim of this more formal presentation of GE is to aid the understanding of the components of the search. One extension of the formal description could be using it for improvement of operators and mapping. The descriptions are inspired by Altenberg [3] where a canonical EA is presented: one addition here is the mapping part, Eq. (9.9). This presentation will be used for the disruption discussion in Chapter 10.

A population is a vector of individuals, where the individuals are defined by their chromosomes C, $\Omega = [C_0, C_1, \dots]$. The difference between populations $\Delta(\Omega, \Omega')$ is created by the different operators, where each is operating on either a single individual, on pairs or on the entire population. Replacement of individuals in a population is denoted by ρ, when two populations, Ω, Ω' are joined and a population, Ω'' is returned:

$$\rho : \langle \mathbb{Z}_{2^m}, \dots, \mathbb{Z}_{2^m} \rangle \times \langle \mathbb{Z}_{2^m}, \dots, \mathbb{Z}_{2^m} \rangle \rightarrow \langle \mathbb{Z}_{2^m}, \dots, \mathbb{Z}_{2^m} \rangle, \rho(\Omega, \Omega') = \Omega'' \tag{9.13}$$

Selection in the population is denoted ς and operates on the fitness values ϕ. This value is used to set the selection probability of the individual. The fitness values belonging to individuals are operated on by ς which takes a fitness value and returns a selected individual.

$$\varsigma : \langle \mathbb{R}, \langle \mathbb{Z}_{2^m}, \dots, \mathbb{Z}_{2^m} \rangle \rangle^M \rightarrow \langle \mathbb{Z}_{2^m}, \dots, \mathbb{Z}_{2^m} \rangle, \varsigma(x) = C \tag{9.14}$$

From Section 3.2 on page 47 the other GE operators are mutation, μ, which takes a codon and returns a codon:

$$\mu : \langle \mathbb{Z}_{2^m}, \dots, \mathbb{Z}_{2^m} \rangle \rightarrow \langle \mathbb{Z}_{2^m}, \dots, \mathbb{Z}_{2^m} \rangle, \mu(C) = C' \tag{9.15}$$

and crossover, ξ which takes a pair of individuals and returns a pair (though it is possible to create crossover operators that only return a single individual)

$$\xi : \langle \mathbb{Z}_{2^m}, \ldots, \mathbb{Z}_{2^m} \rangle \times \langle \mathbb{Z}_{2^m}, \ldots, \mathbb{Z}_{2^m} \rangle \rightarrow [\langle \mathbb{Z}_{2^m}, \ldots, \mathbb{Z}_{2^m} \rangle, \langle \mathbb{Z}_{2^m}, \ldots, \mathbb{Z}_{2^m} \rangle], \xi(C_0, C_1) = [C_0', C_1'] \tag{9.16}$$

The full canonical GE with all the components in the system is created by combining mapping Eq. (9.12), crossover Eq. (9.16), mutation Eq. (9.15), selection Eq. (9.14) and replacement Eq. (9.13).

The aim of this section was to present the different component spaces in GE and how they combine with the grammar mapping in GE. All of these affect the search performance, and require design choices by the implementer. Now we will describe the meta-grammar GE implementation the mGGA.

9.2 The mGGA

The meta-grammar GE implementation, the mGGA, will be investigated here, as well as what is preserved after a change. The entire mapping process is illustrated in Fig. 6.1 on page 101 and the mapping ζ for the mGGA can be broken into two steps, the meta-step $\zeta_m : C_m \rightarrow \zeta_s$ and the solution step $\zeta_s : C_s \rightarrow \Psi$, where the mapping $\zeta_m : C_m \rightarrow G_s \in \Psi_s$ is onto (depending on Ψ). The generation of a solution from the mGGA can be viewed as a derivation in two stages, using solution grammar $G_s = \langle N_s, \Sigma_s, R_s, S_s \rangle$ and meta-grammar $G_m = \langle N_m, \Sigma_m, R_m, S_m \rangle$. The string the mGGA generates is a valid grammar itself, $G_s = L(G_m)$, and $L(G_m) = \{x : S_m \xrightarrow{*} x,\ x \in \Sigma_m^*\}$. This gives $(N_s \cup \Sigma_s) \subseteq \Sigma_m$. Combining Eq. (9.1) and (9.9) and (9.11) for each input codon sequence gives the following expression for mapping and evaluating mGGA (see Fig. 9.2) as in Eq. (9.12), except that the input grammar is generated by Eq. (9.1), (9.3) and (9.5)

$$\phi = h(k(g(f(\langle g(f(\langle G_m, P_m \rangle, C_m)), P_s \rangle, C_s)))) \tag{9.17}$$

The mGGA is created by generating a derivation tree $D_m = D(G_m)$ from the meta-grammar, using the meta-grammar chromosome to determine which production to choose from each rule. Then expand the terminals in the meta-grammar that are non-terminals in the solution grammar (terminals in the meta-grammar that are "quoted" are non-terminals, or BNF syntax in the solution grammar). To select which production to choose from each rule the solution chromosome is used. In other words, a meta-grammar is a more compact description of a large grammar.

Fig. 9.2 shows a visualization of the mGGA algorithm and mapping.

Full mGGA Description

The major difference from standard GE is that mapping Eq. (9.12), crossover Eq. (9.16) and mutation Eq. (9.15) operate on the separate chromosomes. This section has presented the mGGA and the added parameters and operators that it uses for search.

9.3 Discussion

In the process of understanding the mapping from input to output via a CFG some points are raised. Ideally, when considering how a grammar constrains the search space, the grammar is a subset of the solution space $L(G) \subseteq \Psi$ which contains the optimal solutions. However, this property is difficult to guarantee.

Grammar Design

The regularity of the grammar can have obvious effects on the change impacts, by guaranteeing, delaying or making the change redundant. Mixed rules, i.e. rules with both non-terminal and terminal productions, will make the analysis more difficult. Moreover, mixed production choices, i.e. productions with both terminal and non-terminal symbols, further complicate analysis. Therefore, the use of normal forms, e.g. those of Chomsky or Greibach [53], could facilitate analysis, but these forms also impose bias on the search. In

addition, a machine translation might be needed since such normal forms might not be intuitive to write.

9.3.1 The mGGA

In the mGGA, another view of the solution chromosome is that the mapping can be seen as stochastic. This is shown in Tab. 9.1 i.e. no change in input can still change the output, if the solution chromosome is unchanged and the meta-grammar has changed.

The grammar distribution is independent of the derivation order. The genotype is interesting when it is storing information in the genotype and mapping to another representation. Other approaches, e.g. EDAs, manipulate the distribution and use genotypes randomly generated from the distribution instead. Another view is the meta-grammar as a distribution; in this distribution a search for a solution grammar is taking place (this makes it more like an EDA). In the mGGA search is concentrated on some subtrees, through the use of the solution grammar. The parameters of the model (solution grammar) are updated by evolution. This approach is top-down, since the meta-grammar is fixed and no new rules will emerge, only the bias towards them. In addition, the n-sample approach, see Chapter 7 on page 112, brings the mGGA closer to an EDA.

9.4 Summary

This chapter presented a formal description of GE, as well as of the mGGA, aiming to understand the properties of the representation. The different bias in GE have been shown clearly. The behavior of the mGGA in comparison to GE regarding changes in genotype has also been examined. The analysis has revealed more clearly how different grammar designs affect the type and impact of changes. Given our formal description we can now attempt a strict analysis of disruptions in GE. In Chapter 10 disruptions in the phenotype caused by a change in genotype are examined.

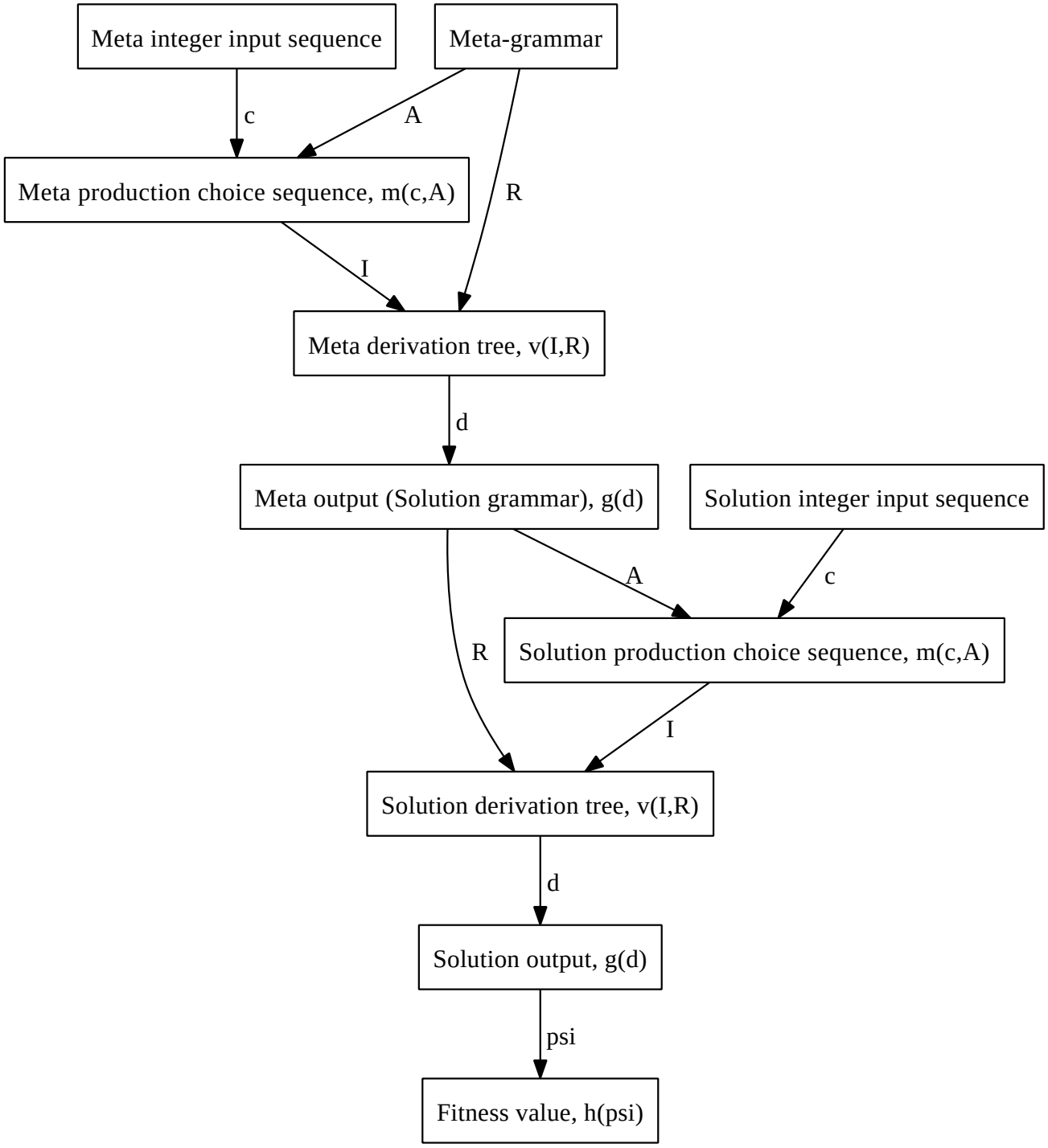


Fig. 9.2: The mapping in the mGGA

Chapter 10

Theory of Disruption in GE

In this chapter we ask the question: what happens to the output (phenotype) when there is a change in the input (genotype) and a CFG mapping is used in GE? The aim of this chapter is to extend the studies of the genotype-to-phenotype mapping and study the effect on derivations and output resulting from one single change in input as well as from multiple changes in input. The mapping of GE introduces redundancies by Eq. (9.3) and dependencies on previous input and the parent in the derivation tree, as well as the design of the grammar. These properties have effects on the preservation of the derivation; often the changes can be quite large, something which in GE is called a "ripple", for the multiple changes that can occur during e.g. mutation or crossover [106]. When the locality of the mapping in GE was investigated [138], the study concluded that some operators in GE had low locality, i.e. genotypic neighbors did not correspond to phenotypic neighbors.

It will be shown that the probability for the phenotype to change increases with the position of the codon. For the mGGA it has been shown that even changes to terminals on the meta-chromosome mapping are disruptive to the production choice context of the solution, since all the terminals and non-terminals disrupt as much as non-terminals would in GE. In addition, the effects of a change on the input will now be defined and labeled. Furthermore, a schema theorem for a version of GE will be formulated. Also, for π-GE it will be seen that the codons which are expanded in the end of a derivation have a higher

probability of their context being changed than the ones expanding early in the derivation.

In Section 10.1 a single change in the input to the grammar is examined. In Section 10.2 we also study the preservation of the derivation after multiple changes. The question of change as an expansion from the new start symbol in the sub-derivation tree, for all unexpanded non-terminals in the derivation, is also raised in Section 10.3. Finally, in Section 10.4 an attempt is made to formulate a schema theorem to quantify disruptions, inspired by Whigham [159].

10.1 Single Change in the Chromosome

The key to understanding changes in GE are the dependencies that occur due to the sequential use of the chromosome and mapping as shown in Fig. 9.1 on page 146. Here we will examine changes in the different stages of the GE mapping.

10.1.1 Codon Change

Consider a single change that could be performed by integer flip mutation of a codon in the codon input sequence and where X is a discreet random variable, $c'_j = X, X \in [0, 2^m - 1]$. This would change the old chromosome C into the new chromosome $C' = \langle c_1, \ldots, c'_j, c_{j+1}, \ldots, c_n \rangle, n = |C|$.

Example 13 (Single codon input change) Consider a single integer flip on a chromosome $C = \langle 42, 666, 13, 49 \rangle$ where the codon at position 1, c_1 changes from 666 to 303, then $C' = \langle 42, 303, 13, 49 \rangle$. □

Let X be a continuous random variable in $[0, 1]$ and p_{mut} the probability to change per codon. The events of codon change and mutation are two separate events. First a mutation event occurs and then the codon value change event, depending on the mutation event. The probability that mutation will happen is $p_{mut} = p(X \leq p_{mut}),\ 0 \leq p_{mut} \leq 1$.

Definition 13 (Codon change) The event that the codon changes when mutation occurs, is the probability of the codon changing multiplied by the mutation probability:

$$p(\text{codon change}) = p(c \neq c')\, p_{mut} \qquad \square$$

Chromosome Change

The probability for a codon to change can be written as $p(c \neq c') = \frac{2^m - 1}{2^m}$. When looking at the entire chromosome the probability of a chromosome to not change is dependent on the chromosome size, the mutation rate and the possible codon values. This probability can be written as:

$$p(C = C') = \prod_{j=0}^{|C|-1} (1 - p_{mut} p(c_j \neq c'_j))$$

10.1.2 Integer Production Choice Change

For the integer production choice sequence we get the following.

Definition 14 (Integer production choice change) The event that the derivation changes when mutation occurs and the changed codon changes the production choice in the input, is the probability of input changing multiplied by the mutation probability:

$$p(\text{production choice change}) = p(i \neq i')\, p(c \neq c')\, p_{mut} \qquad \square$$

This states the fact that the probability of not changing is the product of the mutation probability and the number of productions in the rule: $p(i \neq i') = p_{mut} \frac{|A_k|-1}{|A_k|} \frac{2^m - 1}{2^m}$.

The probability $\mu(n)$ for an integer production choice sequence of length $n \leq |I|$ to not change ($I = I'$) when a uniform mutation rate p_{mut} is applied to the chromosome is the

```
<S>::=<A>|<B>|<C><C>
<A>::=a|b
<B>::=c|d|e|f
<C>::=g|h
```

Grammar 10.1: Grammar for Ex. 14

probability of the mutated codon c'_j selecting the same production. Using Def. 14,

$$\mu = \prod_{j=0}^{n}(1 - p(\text{production choice change})) \quad = \prod_{j=0}^{n}(1 - p_{mut}\, p(i_j \neq i'_j)\, p(c \neq c')) \quad (10.1)$$

The mapping dependency creates the possibility of context changes, i.e. the current rule in the tree changes, which means that the codons are not choosing the same production choices. It is the dependency on previous choices and on the size of the subsequence which gives the possible multiple output changes given a single input change. In CFGs the choice of production is only dependent on the current rule. With a grammar the probability of maintaining a sub derivation begining at zero given a change is $p(i_j = i'_j | i_0 = i'_0, \ldots, i_{j-1} = i'_{j-1}) = \prod_{k=0}^{j} p(i_k = i'_k)$, taken from Def. 6.

Definition 15 (Context change) In a grammar without duplicate rules the context changes if the parent (non-terminal) changes. □

A context change in our notation is $A_k \neq A'_k$. The codon selecting the parent is influenced by the sizes of previous subtrees in the derivation, since changes in subtree sizes will change the codon used for mapping.

It is well worth considering when designing a grammar for GE that mutation affects the probabilities of the input to change. In Ex. 14 there are some examples of changes in the grammar design and of change rates.

Example 14 (Input change) For the input codon sequence $C = \langle 42, 666, 13, 49\rangle$ and mutation rate $p_{mut} = 0.1$ and Grammar 10.1 the input sequence is $I = \langle 2, 0, 1\rangle$ then from Eq. (10.1) the probability that it does not change by mutation is (where

```
<S>::=<A>|<B>|<C><C>
<A>::=a|b
<B>::=c|d
<C>::=e|f|g|h
```

Grammar 10.2: Grammar variant for Ex. 14

$I' = f(\langle G, P\rangle, \mu(C))$, we also approximate $p(c \neq c') \approx 1$):

$$\begin{aligned} p(I = I') &= (1 - 0.1 \cdot 0.66)(1 - 0.1 \cdot 0.5)(1 - 0.1 \cdot 0.5) \\ &= 0.84 \end{aligned}$$

If the mutation rate is increased to $p_{mut} = 0.2$ then:

$$\begin{aligned} p(I = I') &= (1 - 0.2 \cdot 0.66)(1 - 0.2 \cdot 0.5)(1 - 0.2 \cdot 0.5) \\ &= 0.70 \end{aligned}$$

If a different grammar, Grammar 10.2 is used as well then (where $I = \langle 2, 2, 1\rangle$):

$$\begin{aligned} p(I = I') &= (1 - 0.2 \cdot 0.66)(1 - 0.2 \cdot 0.75)(1 - 0.2 \cdot 0.75) \\ &= 0.63 \end{aligned}$$

Note that Grammar 10.1 and 10.2 have the same terminal sets but do not produce the same language. □

Furthermore, Def. 14 gives only an upper bound for no change in the phenotype via mutation, since there are grammars which give the same phenotype even though the production choice sequence has changed. This is an example of redundancy created by the grammar. The probability for the phenotype of the individual not to change when a mutation event has occurred is $0 \leq p(\psi = \psi') \leq \mu$.

One question that has been raised several times regarding GE is that the sequential input and deterministic mapping will create multiple changes in production choice sequence

from one single change in the genome, i.e. a ripple [106, 138] as empirically investigated by Castle and Johnson [21]. The following section helps to show that each element in the derivation has a higher probability of change the further the point of genotypic change is from the root of the derivation tree, since there is a dependency on previous choices and parents.

Derivation change

The probability for no change to occur in an integer production choice sub-sequence decreases as the index of the codons, j, increases. That is, the longer the sequence the higher the probability of change. Where $\mu(n)$ is the probability for an integer production choice sequence of length $n \leq |I|$ to not change. This can be written as $\mu(j-1) \geq \mu(j),\ 1 \leq j \leq |I|$. First, from Eq. (10.1) the probability of changing the sequence is:

$$\mu(j) = \prod_{k=0}^{j} (1 - p_{mut}\, p(i_k \neq i'_k)\, p(c_k \neq c'_k)) \tag{10.2}$$

This gives $\mu(0) \geq \mu(1)$. Since $p_{mut}, p(c_k \neq c'_k), p(i_j \neq i'_j) \leq 1, j \leq |I|$, we can induce that $\mu(j-1) \geq \mu(j)$.

Note that for simplicity the deterministic choices in the grammar are collapsed.

In Ex. 15 the effect on input with higher index in the sequence I is shown.

Example 15 (GE derivation change) For the input codon sequence $C = \langle 44, 666, 13, 49, 606, 303 \rangle$ and mutation rate $p_{mut} = 0.1$ and Grammar 10.3 $f(\langle G, P \rangle, C) = \langle 2, 0, 1, 0, 0, 1 \rangle$. Derivation is shown in Fig. 3.3 on page 41. Then the probability for i_0 to change, from Eq. (10.2), is $\mu(1) = 1 - 0.1 \cdot 0.66 = 0.93$ and for i_3 it is:

$$\begin{aligned} \mu(3) &= (1 - 0.1 \cdot 0.66)(1 - 0.1 \cdot 0.5)(1 - 0.1 \cdot 0.66)(1 - 0.1 \cdot 0.5) \\ &= 0.79 \end{aligned}$$

```
<S>::=<A>|<B>|<C><B><C>
<A>::=a|b
<B>::=c|<D><E>|d
<C>::=g|h
<D>::=i|j
<E>::=k|l|m
```

Grammar 10.3: Grammar for Ex. 15

The probability for the derivation tree root to change, from Eq. (10.2), in δ_0 is 0 and for δ_3:

$$\begin{aligned}\mu(3) &= (1 - 0.1 \cdot 0.66)(1 - 0.1 \cdot 0.66) \\ &= 0.90\end{aligned}$$

□

Another example is the probability to change the symbol corresponding to the root which is lower than the i_j corresponding to any succeeding input, if the selected production i_j has the same number of productions as other non-terminals. Thus, there exists a possibility to reach anywhere in the search space which is less or equal to p_{mut} at the first codon. A note regarding the grammar could be to use a different root in the phenotype as compared to the genotype.

To summarize, the linear input sequence and the CFG make a change at the end of the derivation more probable than changes in the beginning. This section has established how changes in the genotype affect the phenotype for GE and has also given bounds for the disruptions of the derivation.

10.1.3 Change Grammar Design

The contribution of this section is that the studies regarding design of grammars that reduce ripple changes are extended by an analysis of derivation trees and their probability to change the context of the mapping. This section will attempt to extend previous work on grammar design and how it should be informed by a theory of change. First we define

a term in a GE derivation regarding unexpanded non-terminals in the derivation sequence or tree and call them ripple sites.

Definition 16 (Ripple site) Ripple sites are unexpanded non-terminals in the derivation after the current non-terminal. If the derivation $\delta_k = \alpha A \beta, \alpha \in \Sigma^*, A \in N, \beta \in V^*_{N>0}$, where a set containing at least one non-terminal is $V^*_{N>0} = \{x : \exists x \in N, x \in V^*\}$, it contains at least one ripple site. □

We denote the number of ripple sites for derivation $\delta_k = \alpha A \beta$ as $|\beta_N|,\ \beta_N = \{x : x \in N, x \in \beta\}$

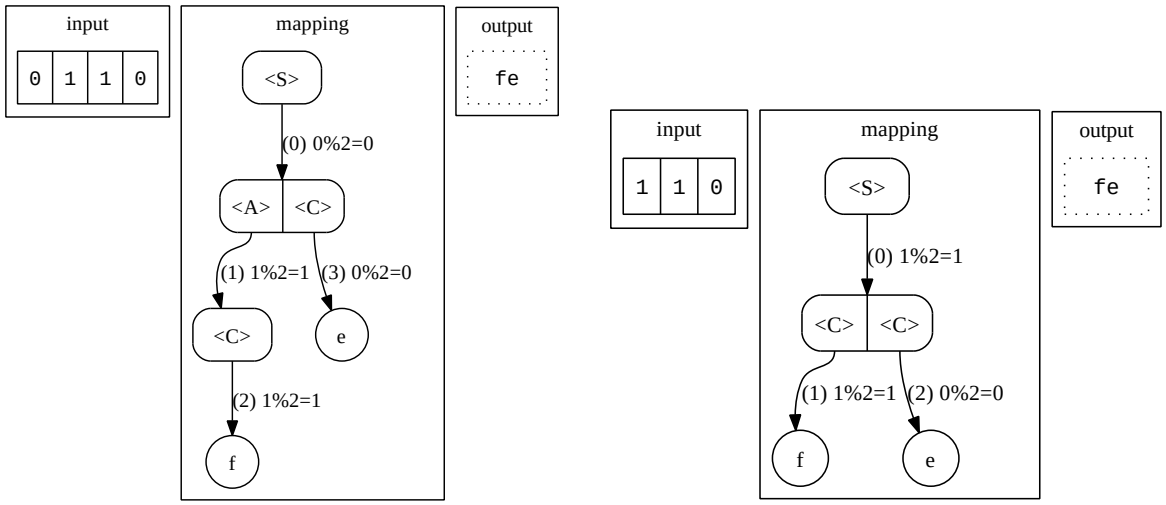
Whigham [159] talks about disruptions and the possibility to reduce the number of non-terminals and rules, in order to facilitate schema propagation in Grammatical Genetic Programming. This is also discussed by Nicolau [103] in a paper where the aim is to reduce the number of such ripple sites in the derivation tree by limiting the number of non-terminals in the grammar, although it is not general and it should be noted that not all grammars are reducible in this way. The use of recursive rules prevents generalization for grammars to produce the same language with a reduced grammar.

For GE the disruption to the input I from terminals and non-terminals can be given a lower bound. The lower bound is related to the number of edges in the derivation tree, which is the number of input production choices. The number of non-terminal productions, $|n|$, in a derivation subtree coming from I, where $|R_\Sigma(D)|$ is the number of inputs, choosing productions with only terminals in the derivation tree, is:

$$|n| = |I| - |R_\Sigma(D)| \tag{10.3}$$

In Ex. 16 there is an example of different grammars for the same language building different derivation trees, thus affecting the degree of change in the derivation tree if the input codon sequence is changed.

Example 16 (CFG designs) This example shows the effect of different numbers of non-terminals. Note that the probabilities for the words generated by the grammar will not be



(a) Original (b) Reduced

Fig. 10.1: Grammars with different numbers of rules

the same in the different grammars. First we have the grammar

```
<S>::=<A><C>|<C><C>
<A>::=<B>|<C>
<B>::=c|d
<C>::=e|f|g
```

with $C = \langle 0, 1, 1, 0 \rangle, I = \langle 0, 1, 1, 0 \rangle, |n| = 2, |I| = 4, |R_\Sigma(D)| = 2$. This can be rewritten as

```
<S>::=ce|cf|cg|de|df|dg|ee|ef|eg|fe|ff|fg|ge|gf|gg
```

with $C = \langle 6 \rangle, I = \langle 6 \rangle, |n| = 1, |I| = 1, |R_\Sigma(D)| = 0$.

```
<S>::=<A><C>|<C><C>
<A>::=c|d
<C>::=e|f|g
```

with $C = \langle 1, 1, 0 \rangle, I = \langle 1, 1, 0 \rangle, |n| = 1, |I| = 3, |R_\Sigma(D)| = 2$. Fig. 10.1 shows that it is the length of the input that is affected by the grammar. □

The Eq. (10.3) gives us a lower bound for the probability to change if there is a change in the chromosome and if a rule with non-terminals in the productions is changed. The probability of changing a non-terminal such that the derivation tree length $|D(i_j)|$ changes is $p(i_j \neq i'_j)(|I| - |R_\Sigma(D)|)$. For GE 1/2 is the highest production choice probability, if there is only one production it is deterministically chosen. Moreover, a lower bound is given if there are no mixed rules, for those production choices that are not only terminal. Thus, a grammar that is as compressed as possible, and which allows for a language with the desired sentences will be the least susceptible to disruption. A caveat is that such a grammar might not be the most intuitive to write.

This section has investigated how grammar design can influence the effect on output of single changes in input and output. Now we turn our attention to multiple changes.

10.2 Multiple Changes in the Chromosome

The single disruption analysis in Section 10.1 is used as a basis for analysis of multiple disruptions. This helps us to understand crossover. A single point crossover can be considered as when after the crossover point multiple changes to the codon occur.

10.2.1 Crossover

A single point crossover at point xo in the input codon sequence, $1 \leq xo \leq min(|C|, |C'|)$ results in $C = \langle c_1, \ldots, c_{xo-1}, c'_{xo}, \ldots, c'_n \rangle$ and $n = |C'|$, $C' = \langle c'_1, \ldots, c'_{xo-1}, c_{xo}, \ldots, c_n \rangle$ and $n = |C|$. Here, only one child from the crossover is considered. Change is considering the parent which contributed with the beginning of the chromosome.

Definition 17 (Crossover change) The probability for a crossover event on the input to change the output can be written as $1 - \xi(xo, n)$:

$$\xi = p_{xo} \prod_{j=xo}^{n} p(i_j = i'_j) \tag{10.4}$$

□

Crossover behavior is similar to mutation but from the crossover point xo to the end of the chromosome C, as regards Eq. (10.4), a longer chromosome makes disruptions more probable. Note the similarity to $p(i_j = i'_j | i_0 = i'_0, \ldots, i_{j-1} = i'_{j-1})$ crossover generating the same sequence as one parent.

After discussing the impact of single and multiple change to the input, it might be worthwhile to see what defines change in the output and whether it can be classified.

10.3 Input Change and Output Preservation

It is known that one change in input can change the order of expansion in the grammar completely, leading to multiple changes in output by changing the size of the subtree rooted in the expansion, as the previous sections have shown. Two types of input changes have been mentioned, single and multiple. The questions we now pose are: can our understanding of the preservation of the derivation after a change be analyzed and classified further than the insight of the "intrinsic polymorphism"[106] of GE? If one codon changes, how much does the context for the succeeding codons change? Here an attempt is made to classify the different possibilities of output preservation. To begin with we investigate those types of change on the input as shown in Section 9.1 on page 147 and the change effects that originate from them.

10.3.1 Change Effects

Given the redundant mapping and parent dependency in GE, a change to the input will be one of two types: change or redundant.

Definition 18 (Redundant change) Redundant changes occur when a change in the codon does not result in a change in derivation $i_j = i'_j, c_i \neq c'_i, I = I', C \neq C'$. □

Of course this also gives $D = D'$.

The change can be direct, the change in input is reflected in the corresponding output, or dependent on the previous mappings. Here it is the grammar and the mapping that

determine the degree of redundancy after a change in the chromosome. A preservation occurs when the input is the same, the codon is the same, but the context (the previous derivations, i.e. the parents) is changed. Thus, a return to the same output with the same input is preservation. This is different from when preservation of output occurs from different input. Tab. 9.3 presents an overview of the types of changes. The point is that each of the indirect or extended changes are artifacts of the GE indirect mapping from a sequence, i.e. from the grammar. Changes to the chromosome are all direct. The indirect encoding of GE allows for indirect changes, i.e. parts not directly connected with the change in the original output are affected by the mapping to the new representation. Both the indirect and direct changes are deterministic, being properties of the encoding.

When any of the changes occur there will be a change effect. See Tab. 10.1 for an overview of the effects of change. The definitions of change effects are based on subtree size $|D(A_k)|$ from where the change occurs and the number of ripple sites $|\beta_N|$ at that point in time (derivation) when we are reading the changed codon. The argument for this definition comes from the fact that changes are based on codon changes and/or parent change. Therefore, if a subtree size changes the codons used to determine the other subtrees at the potential ripple sites, the new derivation subtree size will have changed from the original tree. Thus, this will lead to the possibility of different codons for deciding production choices from the unexpanded non-terminals at the ripple sites.

Some change effects can occur in sequence since the derivation tree has maintained some of its original structure. The change effect occurs as soon as there is a change which is not redundant. A branch effect can be followed by another branch effect. As seen in the definition a branch effect can also be followed by the special case of a branch terminal effect. Within a change effect there can be other change effects. The ripple tail effect is when the tail end of the tree changes. This includes the special case where the root node changes. The effect in ripple contained is the case when a ripple can be limited by canceling out the change in subtree size on the first ripple site by the same size from preceding ripple sites, thus leaving the next site unchanged. That creates a possibility to isolate ripples to subtrees, if they occur at a position with more than 1 ripple site.

Tab. 10.1: Derivation tree change effects for the derivation $\delta = \alpha A \beta, \alpha, \beta \in V^*$ with derivation tree $D(A)$. The change effects are dependent on if the subtree changes size, then it is either a ripple or a branch

Effect	**Subtree size**($\lvert D \rvert$)	**Ripple sites**(β_N)
Ripple (R)	$\lvert D \rvert \neq \lvert D' \rvert, \lvert \beta_N^{RC} \rvert = 0$	$\lvert \beta_N \rvert > 0$
Ripple ripple (RR)	$\lvert D \rvert \neq \lvert D' \rvert, \lvert \beta_N^{RC} \rvert = 0$	$0 < \lvert \beta_N \rvert \leq 1$
Ripple contained (RC)	$\lvert D \rvert \neq \lvert D' \rvert, \lvert \beta_N^{RC} \rvert > 0$	$\lvert \beta_N \rvert \geq 2$
Ripple tail (RT)	$\lvert D \rvert \neq \lvert D' \rvert$	$\lvert \beta_N \rvert = 0$
Branch (B)	$\lvert D \rvert = \lvert D' \rvert$	
Branch terminal (BT)	$\lvert D \rvert = \lvert D' \rvert, \lvert D \rvert = 1$	

When the number of ripple sites is less than 2 the ripple cannot be contained, as with the ripple tail effect. $\lvert \beta_N^{RC} \rvert$ denotes the number of subtrees which cancel each other out, $\beta_N^{RC} = \{\exists n \in [2, \lvert c \rvert] : \sum_{i=1}^{n} \lvert D_i \rvert = \sum_{i=1}^{n} \lvert D_i' \rvert\}$.

These definitions allow some sequences of change effects to be classified. If a branch effect or a branch terminal effect occurs, all of the other change effects can occur afterwards. If a contained ripple effect occurs and the difference in subtree sizes for the ripple sites is 0 and there are ripple sites left, all of the other types of changes can occur.

Preservation of output can occur from two sources, either the context of the input is preserved or a different input generates the same output as was found in the original. To identify the other type of preservation the grammar must be examined.

First, what happens when the derivation changes? Since the sequence is known some inferences about what is preserved can be made. The preservation from the root can be shifted from the site of the initial change before the expansion of the start symbol $S = A_k$, for all the unexpanded non-terminals. A shift refers to when the input has been changed but the output only changes at a subsequent position. The shift can occur with more than one step from where the codon changed. The same applies for crossover but the codons are also changed, which makes preservation less probable.

Example 17 (Change effects) Grammar 10.4

The original individual in Fig. 10.2(a) can for example change after a branch change in Fig. 10.2(b).

```
<S>::=<A>|<B>|<S><C>
<A>::=<B>|<C>
<B>::=c|d
<C>::=e|f|g
```

Grammar 10.4: Grammar for Ex. 17

10.3.2 Branch Change

Subtree sizes are also important for how much the context will change.

Definition 19 (Branch) The change can be contained to a branch. If the change affects $D(A_j)$ and the size is maintained $|D(A_j)| = |D(A'_j)|$. □

This definition leads to all terminal changes being classified as branch changes.

Example 18 (Branch change) The grammar from Ex. 17 is used again Fig. 10.2(b) □

The probability for a branch not to change is the same as Eq. (10.1), but at different intervals and indices:

$$p_m^{|D(i_j)|} \prod_{j=k}^{|D(i_j)|} p(i_k = i'_k), 0 \leq j \leq k, k \leq n$$

The branch changes can be described and be given probabilities to occur. These changes in the derivation interval can occur multiple times, thus changing branches but still maintaining subsequent sequences. The parents have to be the same and the preceding branches must be of the same size. If the size of the subtree is one, $l = 1$ the change is in a terminal. $\pi(i_j)$ is the set of parents for j. Probabilities for a branch are:

$$\prod_{k \in \pi(i_j)} p(i_k = i'_k) \prod_{k \notin \pi(i_j) \cap k < j} p(|D(i_k)| = |D(i'_k)|) \tag{10.5}$$

This can also be written for the derivation, where there is a non-terminal in the derivation at the point of change, $\delta_k = \alpha A \beta, \alpha \in \Sigma^*, A \in N, \beta \in V^*_{N>0}$ and $|D(A_k)| = |D'(A_k)|, D(A) \neq D'(A)$.

Apart from changes that preserve the subsequent subtrees there are changes that can ripple through the entire derivation tree.

10.3.3 Ripple

Definition 20 (Ripple) The context changes if the branch size changes and if there is more than one ripple site. □

Example 19 (Ripple change) The grammar from Ex. 17 is used Fig. 10.2(d). Or with e.g. $\delta_k = \alpha_0 A_0 \alpha_1 A_1 \beta, \alpha \in \Sigma^*, A \in N, \beta \in \Sigma^*$. □

Thus, the codons change mapping in the new context if $c_j \neq c'_j$ and $|D(A_j)| \neq |D(A'_j)|$, where $D(A_j)$ is the sub derivation tree starting with i_j.

For ripple a shift can occur due to the input codon sequence and the grammar. The preserved subtrees appear from each non-terminal expansion of the input codon sequence; it is difficult to speak of a preservation of subtrees since these effects could be purely random.

Example 20 (Ripple Contained) The grammar from Ex. 17 is used Fig. 10.2(c). There is a preservation of the output after a ripple site, even though the subtree where the change occurred has a different size. □

The number of productions can guarantee or prevent context change, with the order of the productions being important. The derivation context comes from the parent $A_k = A'_k$ and will be unchanged as long as $i_j = i'_j$. For $i_j \notin \pi(i_j)$ only $|D(i_j)| = |D(i'_j)|$ needs to hold, meaning there can be a shift even if $|D(i_j)| \neq |D(i'_j)|$. For $|D(i_j)| \neq |D(i'_j)|$ a branch is maintained if $|c_j - c'_j| \bmod |A_k| = 0$ or $c_j = c'_j$, and $|A_k| = |A'_k|$, or has a probability to not change $|c_j - c'_j| \neq 0$ and $|A_k| \neq |A'_k|$. Take into account $p(|c_i - c'_i| \bmod |A_k| = 0) = 1/|A_k|$.

The grammar reduction technique from Nicolau [103] can be further analyzed from the defined change properties. An example of how a grammar used for low disruption for crossover might be more sensitive to mutation is shown in Ex. 21.

Example 21 (Reduced grammar) The following grammar will exchange material in the same context from crossover events and create a branch change from the crossover point.

```
<R>::=a|b|<R>a|<R>c
```

And mutation will always only change one production, thus using Eq. (10.3) the change is either single terminal change (subtree size 1), expansion or contraction. □

This section has labeled different types of output changes that can occur when the input changes and the representation are projected from a CFG. Showing that there are different ripples depending on how many ripples sites there are, this gives some insight into how context is preserved in GE. In Section 10.4 the focus is turned to how structures in the population are preserved over time.

10.4 Disruption in a GE Population

One often wonders how EC functions. It is often argued, supported by a schema theorem, that the reason why GAs work is that small fit parts of the genotype are propagated during the evolutionary process [46]. This schema theorem has been studied for many different EA systems.

Poli and Stephens [128] generalize the building block hypothesis for variable-length strings and program trees. Whigham [159] investigates a schema theorem for CFGs.

Altenberg [2] discusses evolvability, i.e. the ability of the genetic operator/representation to produce offspring fitter than their parents. He claims that a building block in GA is defined by a correlation between parent and offspring fitness under recombination and that this would also give the desired properties of a genetic operator. Altenberg [3] further tries to generalize the GA schema theorem with Price's theorem. Price's theorem regards the covariance between parental fitness and offspring traits for informing how selection drives evolution.

GE Schema

A schema is a template containing zero or more non-terminals, matching multiple valid individuals. Following Whigham [159], an attempt to describe how structures are propagated during evolution is made in this section. When a derivation δ represents a schema, a "do not care" symbol, often *, used in GA and GP schemata, is not needed, since a sentential form is a valid derivation [161]. The derivation string is expressive enough, where non-terminals denote unexpanded subtrees. From Whigham [159]:

Definition 21 (Schema in CFG) A schema H for a Context-Free Grammar is the sub derivation tree $H = \{x \xrightarrow{*} \beta\}, x \in N, \beta \in V^*$ □

In the schema, $|H|$ is the length of the schema i.e. subderivation tree. H is rooted in a non-terminal and the schema H can occur more than once in the derivation D. Also worth noting is that the individuals that match the schema can differ in size. The number of fixed symbols in a schema is called the order of the schema.

Example 22 (Derivation string schema) The schema for `c<C>e` is shown in Fig. 10.3 on page 186.

A proposed theorem for schema disruption in GE is given next. By studying the probability of a schema to not be disrupted during a generation. This means that the probability of crossover and mutation to change the schema are studied, as well as the probability of selecting a schema using fitness proportional selection.

Schema disruption due to crossover

The probability of a derivation tree D containing H to not change due to crossover is:

$$p(H = H') = \prod_{j=xo}^{h_e} p(i_j = i'_j) \tag{10.6}$$

where h_e is the index in D where H ends and the crossover point $xo, 0 \leq xo \leq |H|$. The probability to change the schema is $\kappa = 1 - p(H = H')$, see Eq. (10.4). The comparison is

made with the parent which contains the start of the chromosome.

Note that cases of redundant grammars are ignored here. Moreover, with the change effect "branch" where the change occurs in the unexpanded non-terminal region the schema will still be maintained.

Schema disruption due to mutation

The probability of a derivation tree D containing H to not change due to mutation is:

$$p(H = H') = \prod_{j=m_s}^{h_e} p(i_j = i'_j) \tag{10.7}$$

With m the point of mutation and $0 \leq m \leq |H|$. The probability to change is $\varpi = 1 - p(H = H')$, see Eq. (10.1).

We define $|D_H|$ as the number of occurrences of schema H in the derivation tree D and the average disruption due to crossover $\overline{K}$ and mutation $\overline{M}$, i.e. the number of schema in each individual in the population times the probability of disruption due to crossover or mutation divided by the total number of schema in the population. This can be written as: (where Ω is the population)

$$\overline{K} = \frac{\sum_{D \in \Omega} \kappa |D_H|}{\sum_{D \in \Omega} |D_H|} \tag{10.8}$$

$$\overline{M} = \frac{\sum_{D \in \Omega} \varpi |D_H|}{\sum_{D \in \Omega} |D_H|} \tag{10.9}$$

The fitness of derivation D is f_D and the average schema fitness for fitness proportional selection and replacement ρ is:

$$\rho = \frac{\overline{f}_H}{\overline{f}} \tag{10.10}$$

$$\overline{f}_H = \frac{\sum_{D \in \Omega} f_D |D_H|}{\sum_{D \in \Omega} |D_H|} \tag{10.11}$$

$$\overline{f} = \frac{\sum_{H \in \Omega} \overline{f}_H}{|\Omega|_H} \tag{10.12}$$

Where $\overline{f}_H$ is average fitness of a schema in a population and $\overline{f}$ is the average fitness of H in the entire population.

Combining Eq. (10.9), Eq. (10.8) and Eq. (10.10) it is possible to set up bounds for the propagation of schema H due to crossover, mutation, and replacement and selection over time. That is, a schema is propagated if it is not disrupted by mutation and crossover and fit enough to be selected by proportional selection. We use p_{xo} as the probability of crossover and p_{mut} as the probability of mutation. $(1-(1-p_{mut})^{|H|})$ denotes the probability that a mutation occurs within the derivation tree needed to describe the schema.

Theorem 1 *The probability $p(H,t)$ of schema H to be found at time t:*

$$p(H,t) \geq p(H,t-1) \cdot \rho \cdot (1 - p_{xo}\overline{\kappa}) \cdot (1 - (1 - (1 - p_{mut})^{|H|})\overline{M})) \tag{10.13}$$

□

PROOF From Eq. (10.10), Eq. (10.8) and Eq. (10.9). ■

Note that a lower bound for schema disruption is the probability that a random derivation gives the schema $p(H) = \prod_{i=0}^{|H|} p(H_i)$, thus reoccurring by random chance. That is a crossover or muation reintroduces the schema. This is important since it shows the impact of the grammar for allowing schema.

In Section 10.3 it was shown what could happen after a change, making crossover more disruptive than mutation, because to avoid disruption there would be a need for many changes, either ripple contained or branch changes. Furthermore, this shows that large schemas are more easily disrupted. It also highlights the role of redundancy in the mapping for maintaining subtrees.

To summarize: this section showed that the probability that an integer element i_j at position j in a derivation sequence will be disrupted will increase when j is increased, both for single and multiple changes.

10.5 The mGGA

The disruption of the mGGA is investigated and what is preserved after a change.

10.5.1 Changes in the mGGA

In comparison to standard GE where only mutations of N can affect the following codons the meta-grammar is different since both Σ and N will affect the generation of the solution grammar and by extension the solution itself. Thus Eq. (10.2) shows that a C_m is very likely to cause context change in the solution given by the solution grammar.

The homologous crossover using two chromosomes is similar to using a two point crossover on one long chromosome with non-uniform points. In the mGGA the setup of using two chromosomes is similar to using one long chromosome with the same crossover rate. Likewise, mapping using two chromosomes is similar to mapping with one chromosome and then at some point in the mapping letting the expansion index restart from the beginning of the derivation, $A_k = A_0$, and changing grammar $G = G_S$ by changing some terminals to non-terminals. With this in mind the change in an individual's phenotype can be described by the change in the genotype of both chromosomes. For the mGGA the probability of a context change can be even higher, since the meta-grammar terminals are non-terminals in the solution grammar. With the solution being dependent on both the meta-grammar and solution chromosomes the length $|C|$ can be subdivided into $|C| = |C_m| + |C_s|$.

10.6 Discussion

In the process of understanding the mapping from input to output via a CFG some points are raised. In this section the discussion will touch on the concept of change measures, expansion order and grammar distributions.

From the definitions of change effects we see that the non-terminal distribution of I is important, especially for one-point-crossover. With this terminology a context-sensitive

operator [50] would guarantee a branch change and insert or replace a subtree so that everything in I' would be within the same context as in I.

From experiments regarding the positional effect of crossover and mutation in GE there is support for the idea that events occurring in the first position of the genotype are more destructive, but they can also be the most constructive, regarding fitness [21]. The ripple effect means that a few changes to the chromosome can be propagated to multiple changes to the phenotype. This affects the local search capabilities of GE and the causality principle, i.e. small changes in input should correspond to small changes in output. Here the bias of the grammar mapping and population effects can help enforce this principle.

10.6.1 *pi*-GE

In Section 9.1 we noted that the function $m(c_j, A_k)$ for choosing which production to expand is dependent on two variables, the current codon c_j and the current rule. In Eq. (9.5) $i = 0$ could be changed to $0 \leq i \leq |\delta_k|$ which would change the order of non-terminal expansion (the left-to-right expansion keeps subtrees adjacent in the I representation). For example π-GE [104, 37] experiments with evolving the expansion order of expressions and the experimental evidence suggests that it works well. The expansion is chosen by using parts of the chromosome to decide which non-terminal to pick for expansion of the number of non-terminals N in the current derivation δ_k,

$$
\begin{aligned}
C &= \langle\langle c_1, \eta_1\rangle, \dots\rangle \\
A_k &= \eta_k(\mathrm{mod}|\nu_{k-1}|) \\
\nu_k &= \{x_i : 1 < i < \delta_k, x_i \in \delta_k, x_i \in N\}
\end{aligned}
$$

Other examples of derivation order could be breadth-first or even a mix of breadth- and depth-first. Variations on this theme were also explored in Hemberg et al. [58]; by changing the derivation order, parts of the genotype can increase the probability of subtrees being preserved by being expanded early, see Eq. (10.2).

The mapping also shows that the probability for derivation tree nodes (i_j, i_{j+n}) to be kept in context depends on n, the size of the subtree at index j. This can be one of the reasons why π-GE [104] might be more successful, as important expansions in the derivation can be shifted towards the beginning of the input, and the shift of the expansion to the beginning of the sequence reduces the probability of disruption for the derivation subtree.

More arguments for π-GE's ability to overcome declarative bias in the grammar by having a more adaptive mapping can be seen in Tab. 9.3 on page 153. The ability to change which non-terminal to expand can mean that mutation creates an immediate or delayed change in the derivation.

10.6.2 GE Schema

The propagation of schemas is bounded by the operators and the expansion of non-terminals. One view of GE is that the integer sequence I tends to converge from left to right, since the schemas are most likely kept at the beginning of the derivation.

For context-preserving operators for mutation and crossover, as well as different replacement methods the schema preservation κ, ϖ, ρ will change, making Eq. (10.6) more similar to Whigham's [159] theorem. These observations might give further support to the need for operators that can perform a more "local" search as well as to the need for a different bias for the schema propagation in new individuals.

Another part of the GE giving rise to schemas is the probability of derivations from N which can contain schemas. This means the re-introduction of schemas due to the bias in the grammar. One view of how the GE works is that it is the probabilities in ζ_s, from random sampling, in conjunction with the evolutionary operators that preserve material. Altenberg [3] also elaborates on the search bias defined by Price's theorem. This shows how a correctly biased grammar increases the average fitness of the populations. Domain knowledge in the grammar also constrains the search.

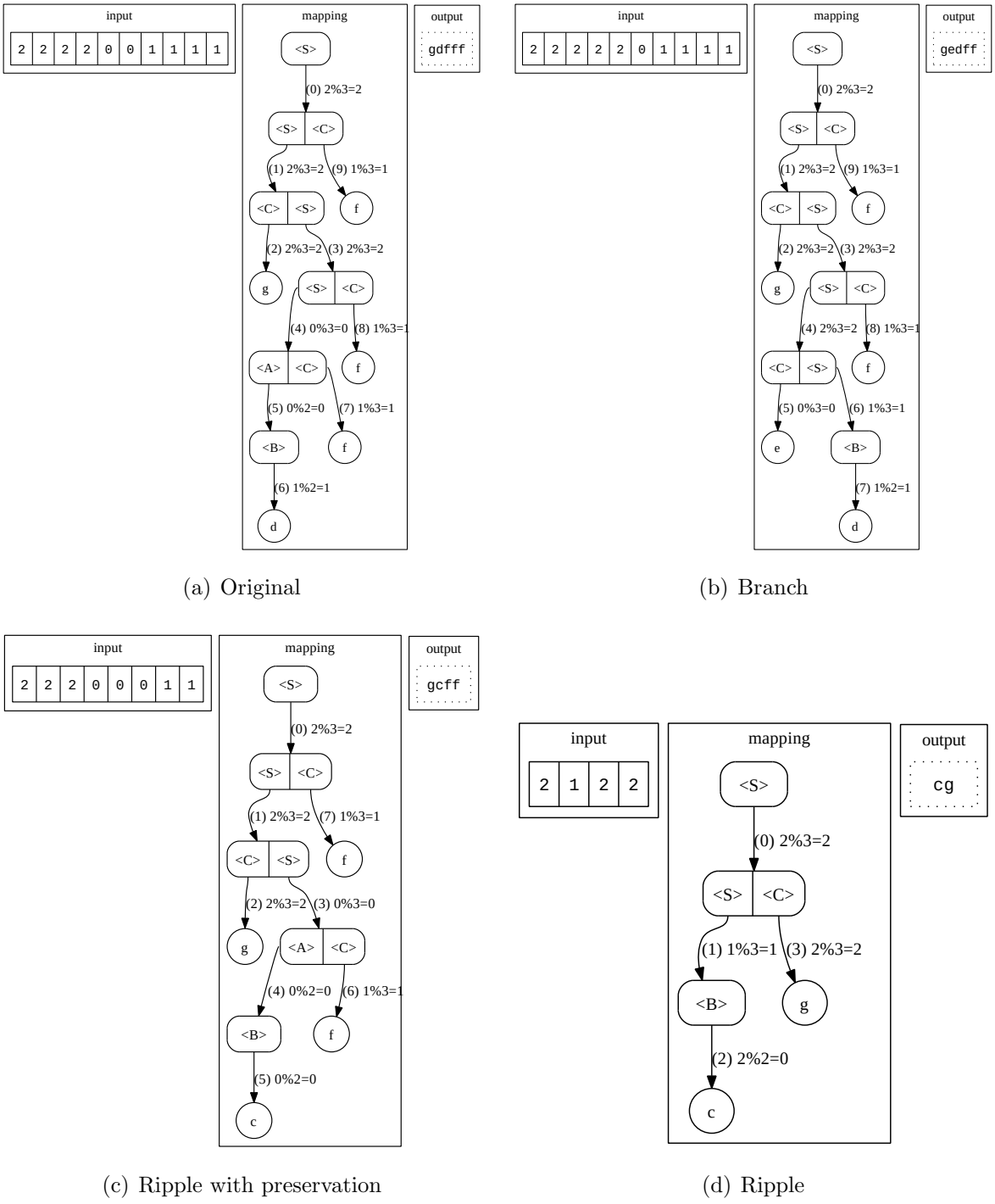
To conclude: the discussion of CFGs for representation have suggested measures for change, different expansion orders and views on grammar distributions.

10.7 Summary

Disruptions in the phenotype caused by a change in genotype have been examined. It has also been revealed that the probability for the phenotype to change increases with the position of the codon in the chromosome. The behavior of the mGGA in comparison to GE regarding changes in genotype has also been examined, and it has been shown that even changes to terminals on the meta-chromosome mapping are disruptive to the solution production choice context. This is due to the dependence of the solution on the grammar generated by the meta-chromosome, i.e. all the terminals and non-terminals disrupt as much as non-terminals would in GE. In addition, the study of changes has revealed a lower bound for the disruption probability in a grammar and thus given some pointers for grammar design, that is, the fewer non-terminals it contains, the less susceptible it will be to disruption. The effects of a change on the input were defined and labeled. Furthermore, a schema theorem for a version of GE has been formulated.

The analysis has revealed more clearly how different grammar designs affect the impact of changes. In addition, the understanding of locality in GE can be enhanced by studying the change effects. Also, for π-GE the analysis of how the mapping works can be aided by the analysis here. It has been shown that the codons which are expanded at the end of a derivation have a higher probability of changing context than the ones expanding early in the derivation.

After this analysis of changes we now study how to measure grammars. In Chapter 11 different measurements of Grammars in GE are investigated.



(a) Original

(b) Branch

(c) Ripple with preservation

(d) Ripple

Fig. 10.2: Example individual and different changes

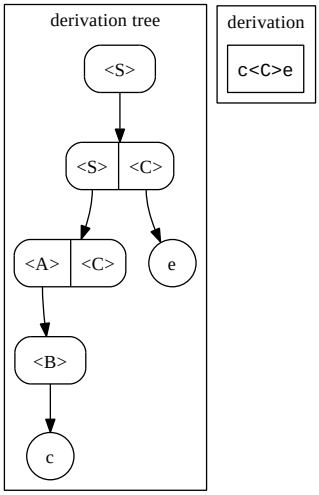


Fig. 10.3: Derivation tree, string schema

Chapter 11

Grammar Measurements

When exploring grammars in GE it is useful to have measures for comparing grammars, thus adding a quantitative element to the intuitive and qualitative aspects of a grammar. The aim of this chapter is to statically analyze grammars. Examples of how to apply this to GE are provided. Measurements of grammars can inform which constraints the grammar imposes on the search, something which otherwise might be difficult for the practitioner to identify. An analysis should include identifying and measuring grammars in order to allow comparison of grammars evolved during the run of the mGGA. One of the aims of this, for example, would be to aid the selection of interesting grammars to investigate, by doing a static measurement before sampling the grammars.

This chapter introduces some properties for grammars in GE, e.g. how many words a grammar can produce and the probability of deriving them. As an example of measures a matrix representation for a PCFG for determining if a derivation would converge, as well as for calculating the expected derivation- and word length will be provided. Finally, there is an analysis of the fitness and creation of operators using the Expected Derivation Length (EDL).

In Section 11.1 the identity and properties of grammars are identified. In Section 11.2 there is a description of how to represent PCFGs in matrices. The measures available from the matrix representation of the grammar are applied to the mGGA in Section 11.3.

11.1 Grammar Identity and Properties

In order to statically analyze a grammar some desirable properties need to be identified. One goal would be to see if there is some information to be had about the search space before sampling from the grammar, i.e. is it worthwhile to explore the grammar. One view would be to look at the grammar and at how many different sentences the grammar can produce, hereby especially keeping in mind the observation regarding the few degrees of freedom in the solution grammars evolved in Chapter 5, 6, 7 and 8. The evaluation of the measures from the grammar can be done automatically as well as manually. For meta-grammars automating the comparison would be preferable, and could involve adding an extra objective to the fitness function basically constraining the search space.

Grammar Properties in GE

The grammar is used to constrain and bias the search for solutions. Some measures might be binary, since a sensible scale might be difficult to obtain. The grammar should guide the search, but can easily be too constrained [130].

One initial division line could be if the grammar would have uniform bias and allow the EA to handle the search, or if more specific bias could aid the search, since the search space can be fairly large. For example, in the mGGA evolution alone is used to search for the grammar bias and structure. Desirable properties for a grammar to be examined could include:

- Reachability of strings in a language, i.e. which words can be derived and their probability, which addresses the correctness of the bias, expressability (e.g. number of paths in the grammar[1]), expected derivation length and number of unique paths
- Probability of reaching strings in a language, which addresses the strength of the bias.

[1]This only works on non-recursive grammars, unless a depth limit is imposed. But since most EA implementations are in some sense bounded by hardware and software specifications or user-defined parameters, the derivations are finite

When comparing solution grammars one could use the unique production ids and take the string edit distance. This will tell you the grammar distances in the population, but not how expressive they are.

11.1.1 Grammar Measurements for GE

As the grammar can have quite an impact on search performance it is not only of theoretical interest to measure grammar properties. The measurements should ideally be able to predict changes in grammars used in GE and in observed outcomes. A grammar measurement is of interest to the practitioner not only when deciding which grammar to use before starting the GE run. The grammar measurements can be used for more than the initial part of the run, they can adaptively change the population during the run. This is especially important since one of the key issues when altering GE is to decide what parts of the algorithm to change.

When using a grammar it is interesting to see how much a change in the probability to choose a production will affect the search properties of the grammar, as well as changes in non-terminals, terminals and rules.

With desirable properties identified for the grammar one might also want to measure the impact of the grammar (distribution) on the search. This is a sample of measures that are computable, some apply to the grammar or individual while others are in the population-, generation- or run-context:

- Number of rules, non-terminals, terminals. This is simple to calculate and can give a very rough estimate of grammar complexity.

- The reachability of strings when the population is using meta-grammars can be checked by examining the string output probability $p(s)$, for the population $p(s, \Omega) = \prod_{i=0}^{|\Omega|} p_i(s), s \in \Sigma^*, 0 \leq p_s \leq 1$.

- String fitness probability, e.g. $\iota = p(s)\phi(s), \phi \in \mathbb{R}$, population probability $I = \prod_{i=0}^{n} p_i(s)\phi_i$ Here it can be difficult to know exactly which string fitness probabil-

ities ι are useful. Therefore, both high and lower values of ι can be useful indicators. High values should indicate high probability of string generation and high fitness and lower values are for low fitness and/or low probability of sentence generation. Added information can be found when studying progression over generations.

After presenting grammar measurements for GE we will now further investigate grammar design in GE and complexity measures.

11.1.2 GE Grammar Design and Complexity Measure

This section aims to further analyze the measurement of grammar complexity from Section 10.1.3. For GE the reduction of a grammar from multiple terminals to only one non-terminal will change the bias because of the uniform production choice probability. Moreover, duplicated production choices will need to be added to maintain the original bias. An alternative to overcome this could be merely to alter the probabilities P, and to allow them to be non-uniform (an additional mapping step $r : \mathbb{Z}_{2^l} \rightarrow \mathbb{R}, r(C) = y$ for mGGA would be necessary).

It can be seen that the probability for derivation tree neighboring nodes (i_j, i_{j+n}) to not change depends on the size of the subtree of the expressions n, see Chapter 10 on page 162. It must be taken into account that multiple non-terminals in a production might be related. For GE, the context will be kept if all the preceding subtree sizes are kept, and not only one.

One question raised by Nicolau [103], is the definition of the complexity of the grammar, there it is referred to as the number of rules. Another definition can be the number of rules and productions, this would allow a distinction between CFGs that produce the same language but with different probabilities, see Ex. 23.

Example 23 (Grammar complexity) Grammar complexity as the total number of symbols on the right-hand side is $c_{RHS}(G) = y, y \in \mathbb{N}$ and as the total number of symbols in grammar $c_{tot}(G) = y, y \in \mathbb{N}$. The grammar G

```
<S>::=<B>|<C>|<D>
<B>::=a|b
<C>::=c
<D>::=d|e|f
```

with $|N| = 4, |R| = 8, |\Sigma| = 6, c_{RHS}(G) = 20, c_{tot}(G) = 44$. This can be rewritten to G_0 but with probabilities not the same as in G.

```
<S>::=a|b|c|d|e|f
```

with $|N| = 1, |R| = 6, |\Sigma| = 6, c_{RHS}(G_0) = 11, c_{tot} = 17$. This can be rewritten as G_1 with the same probabilities as in G.

```
<S>::=a|a|a|b|b|b|c|c|c|c|c|c|d|d|e|e|f|f
```

with $|N| = 1, |R| = 18, |\Sigma| = 6, c_{RHS}(G_0) = 29, c_{tot} = 35$. □

To summarize, we have listed some desirable grammar properties as well as measures for impact of the grammar. Now we will look at a concrete example by using a matrix representation for the mGGA.

11.2 Probabilistic Context-Free Grammars in a Matrix Representation

EC has previously been analyzed to achieve finite Markov chain results [139]. In this section we take a different approach and study the generation of strings from a grammar instead. In this section a matrix representation is used for the mGGA grammar, as was done for PCFGs by Wetherell [157]. One of the aims with this approach is to understand the grammar, which strings are reachable and expressable. Another aim is how the grammars can be compared. The mGGA and its generation of solution grammars highlight grammar generation. Another aim is to be able to compare grammars, this is approached with examples from the mGGA, by using computable intensive measures to investigate bias.

```
<bitstring> ::=  <bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4>
<bbk4> ::= 1 1 0 0
         | 0 0 <bit> 1
<bit> ::= 1
```

Grammar 11.1: Grammar bit string example, again

$$\begin{aligned}
\texttt{<bitstring>} &= r_{00} = p_{00} = P_0 = 1 \\
\texttt{<bbk4>}_0 &= r_{10} = p_{10} = P_1 = 0.5 \\
\texttt{<bbk4>}_1 &= r_{11} = p_{11} = P_2 = 0.5 \\
\texttt{<bit>} &= r_{20} = p_{20} = P_3 = 1
\end{aligned}$$

Fig. 11.1: Non-terminals, rule indexes, probability indexes, unique probabilities and production choice probability

For the mGGA, these measures could be used as an extra objective, both when selecting which solution grammars to explore, e.g. does the solution grammar express many different strings or is it strongly biased to a few, as well as when choosing meta-grammars. The comparison of grammars gives:

- a different view of the probabilities of the productions
- an overview of what expansions can lead to what non-terminal path (Section 11.2.1)
- an overview of which expansions could lead to which symbols (Section 11.2.2)
- information on the production probabilities, e.g. Grammar Defined Introns [116]

The grammar used to explain the matrix representation, Grammar 11.1 is the same as in Grammar 7.1 on page 119. with the rule set R as in Fig. 11.1.

11.2.1 Non-terminal Expectation Matrix

With the help of Wetherell [157] one can define a matrix representation for grammars. When studying PCFGs one question to ask is how many non-terminals are expected in a

string after a non-terminal is rewritten once. The following section will present how this can be written in matrix form. This is achieved by creating a matrix for the production probabilities for each non-terminal and multiply it with the number of occurrences of non-terminal symbols for each production. For Grammar 11.1 this would yield a matrix called $A, 0 \leq a_{ij}, a_{ij} \in \mathbb{R}$, where a_{ij} denotes how many non-terminal symbols of the type in column j are expected after expanding the non-terminal in row i. That would look like:

$$A = \begin{array}{c} \\ \texttt{<bitstring>} \\ \texttt{<bbk4>} \\ \texttt{<bit>} \end{array} \begin{array}{c} \begin{array}{ccc} \texttt{<bitstring>} & \texttt{<bbk4>} & \texttt{<bit>} \end{array} \\ \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \end{array}$$

Note, we do not study the output state of the grammar here, which could also be done. In Grammar 11.1 every time `<bbk4>` is rewritten expected to be expanded to `<bit>` half of the times. A can be derived by multiplying the probabilities for choosing a rule(the LHS) and the number of occurrences of the non-terminal on the right-hand side.

To generate A according to Wetherell [157] Setup a matrix Q for the production probabilities for the non-terminal to have $|N|$ rows indexed by n_i and $|R|$ columns indexed by P_j, element q_{ij} has value p_j if production P_j has non-terminal n_i on its left-hand side and otherwise 0, e.g. for grammar in Grammar 11.1 (`<bbk4>`$_0$ refers to the first production choice in rule `<bbk4>` and `<bbk4>`$_0$ to the second production choice)

$$Q = \begin{array}{c} \\ \texttt{<bitstring>} \\ \texttt{<bbk4>} \\ \texttt{<bit>} \end{array} \begin{array}{c} \begin{array}{cccc} \texttt{<bitstring>} & \texttt{<bbk4>}_0 & \texttt{<bbk4>}_1 & \texttt{<bit>} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Matrix C representing the number of non-terminals per production has $|R|$ rows indexed by productions and $|N|$ rows indexed by non-terminals. Element c_{ij} has a value which is the

number of occurrences of n_i on the right-hand side of P_j, e.g. for grammar in Grammar 11.1

$$C = \begin{array}{c} \\ \texttt{<bitstring>} \\ \texttt{<bbk4>}_0 \\ \texttt{<bbk4>}_1 \\ \texttt{<bit>} \end{array} \begin{array}{c} \begin{array}{ccc} \texttt{<bitstring>} & \texttt{<bbk4>} & \texttt{<bit>} \end{array} \\ \begin{pmatrix} 0 & 8 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

A stochastic expectation matrix for non-terminals, where the elements can be interpreted as the expected number of times a non-terminal will occur when a non-terminal is rewritten once. It is a square matrix $A = Q \cdot C$ with $|N|$ rows and columns and a_{ij} can be interpreted as the expected number of times n_j will occur when n_i is rewritten once. To ensure that a probabilistic grammar is consistent the largest eigenvalue must be less than one[2]. For Grammar 11.1

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 8 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 8 & 0 \\ 0 & 0 & 0.5 \\ 0 & 0 & 0 \end{pmatrix} \end{aligned}$$

This gives the eigenvalues λ for A as $\lambda = [0, 0, 0]$. If A is consistent, i.e. the productions are distributed to the terminal strings and the derivation finishes, $\sum_{w \in L(G)} p(w) = 1$ [157]. It is possible to calculate the non-terminal expectation matrix. It is worth noting that the consistency measure can be used as a binary indicator for grammars supplied, to test if

[2]This is the spectral radius, $\rho(A) < 1$. This means that the convergence of the power sequence of the matrix goes to 0, $\lim_{k \to \infty} A^k = 0 \Leftrightarrow \rho(A) < 1$ [157]

the strings generated will converge to terminal sentences, as well as for testing a range of probabilities for a grammar and seeing when it will converge. The consistency is calculated by looking at the geometric series of A (I is the identity matrix)

$$A^{\infty} = \sum_{i=0}^{\infty} A^i = I(I - A)^{-1} \tag{11.1}$$

Note that in GE there might be a difference between the asymptotic behavior of A^{∞} compared to the finite limit imposed by the implementation. This is a crucial point and good tools are needed to determine the convergence of the series in Eq. (11.1). Eq. (11.1) gives for Grammar 11.1

$$A^{\infty} = \begin{pmatrix} 1 & 8 & 4 \\ 0 & 1 & 0.5 \\ 0 & 0 & 1 \end{pmatrix}$$

This also gives us the possibility to calculate the expected derivation length for a given non-terminal (EDL(n_i)). That is, for a derivation beginning with a non-terminal n_i and ending in a terminal string. The calculation is done by summing all the the elements of the n_i-th row. In the grammar in Grammar 11.1 the EDL for all non-terminals is acquired by multiplying A^{∞} with a column vector of ones.

$$\text{EDL} = \begin{pmatrix} 1 & 8 & 4 \\ 0 & 1 & 0.5 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 13 \\ 1.5 \\ 1 \end{pmatrix}$$

Thus, from an arbitrary derivation starting from `<bitstring>` four occurrences of `<bit>` non-terminals are expected and 13 derivation steps are taken before the derivation is complete. Similarly, `<bbk4>` would be expected to require 1.5 steps and not surprisingly, given no choice, `<bit>` is expected to require a single step.

11.2.2 Terminal Expectation Matrix

Now that we have an expectation matrix for the non-terminals it is also possible to calculate one for the terminals. For Grammar 11.1 on page 192 terminals[3] are defined as $\Sigma = \{\texttt{1}, \texttt{00}, \texttt{1100}\}$. With this information we can calculate the expected word length (EWL).

For Σ a matrix T is defined as having $|R|$ rows indexed by P_i and $|\Sigma|$ columns indexed by σ_j. Element t_{ij} has a value which is the number of times terminal σ_j occurs on the right-hand side of production $|P_i|$, e.g. for grammar in Grammar 11.1

$$T = \begin{array}{l} \\ \texttt{<bitstring>} \\ \texttt{<bbk4>}_0 \\ \texttt{<bbk4>}_1 \\ \texttt{<bit>} \end{array} \begin{array}{c} \begin{array}{ccc} \texttt{1} & \texttt{00} & \texttt{1100} \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \end{array}$$

The expected terminal matrix $W = Q \cdot T$ has $|N|$ rows indexed by n_i and $|\Sigma|$ columns indexed by σ_j. From the example

$$\begin{aligned} W &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \\ 1 & 0 & 0 \end{pmatrix} \end{aligned}$$

Also, here an expected word length can be calculated starting from a given non-terminal

[3]The terminal set could be defined as $\Sigma = \{\texttt{1}, \texttt{0}\}$, but for some problems the positions and combinations of terminals are important.

(EWL(n_i)). That is, as the number of terminal symbols in a word derived from n_i. The terminal expectation matrix $\mathcal{W} = A^{\infty} \cdot W$ has $|N|$ rows indexed by n_i and $|\Sigma|$ columns indexed by σ_j, and is derived as EDL. As an example

$$\mathcal{W} = \begin{pmatrix} 1 & 8 & 4 \\ 0 & 1 & 0.5 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 8 & 4 & 4 \\ 1 & 0.5 & 0.5 \\ 1 & 0 & 0 \end{pmatrix}$$

$$\text{EWL} = \begin{pmatrix} 8 & 4 & 4 \\ 1 & 0.5 & 0.5 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 16 \\ 2 \\ 1 \end{pmatrix}$$

In Section 11.3 an example of this representation and the mGGA will be given.

11.3 Examples of the mGGA in Matrix Representation

We can now calculate the EWL and EDL for grammars. Now we turn to the mGGA to analyze the usefulness of the matrix representation. This section shows how to rewrite grammars used in GE and presents examples of the mGGA.

11.3.1 Rewriting GE Grammars

In order to apply the theory from Section 11.2 to GE, the codon use of GE needs to be taken into consideration. In the mGGA EDL is an upper estimate of the expected codons used, this is because no codon is used if the input is deterministic. The grammar could be rewritten with all the deterministic choices collapsed. This would make a difference for the depth of the new derivation tree compared to the derivation tree of the original

```
<bitstring> ::=  <bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4><bbk4>
<bbk4> ::= 1 1 0 0
         | 0 0 1 1
```

Grammar 11.2: Simplified grammar in Grammar 11.1. The simplification affects the size of the derivation tree

grammar, but for the number of codons used it would make no difference. For the derivation tree $d = f(\langle G, P\rangle, C)$ and $d' = f(\langle G', P'\rangle, C)$, $g(d) = g(d')$ $\text{depth}(d) \neq \text{depth}(d')$, the production probabilities for the rules are the same.

Example 24 (Rewriting GE grammars) The grammar in Grammar 11.1 could be written as in Grammar 11.2. This reduces the derivation tree size and does not affect the codon use.

The grammar for Grammar 11.2 $N = \{\texttt{<bitstring>}, \texttt{<bbk4>}\}$

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0.5 & 0.5 \end{pmatrix} \begin{pmatrix} 0 & 8 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 8 \\ 0 & 0 \end{pmatrix} \qquad , A^{\infty} = \begin{pmatrix} 1 & 8 \\ 0 & 1 \end{pmatrix}, \text{EDL} = \begin{pmatrix} 9 \\ 1 \end{pmatrix}$$

And $\Sigma = \{\texttt{1100}, \texttt{0011}\}$

$$W = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0.5 & 0.5 \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0.5 & 0.5 \end{pmatrix} \qquad , \text{EWL} = \begin{pmatrix} 8 \\ 1 \end{pmatrix} \qquad \square$$

Note that the expected word length could be changed if different symbols are considered as belonging to Σ, e.g. only the characters $0, 1$ could be said to be symbols, $\Sigma = \{0, 1\}$, thus giving an exact length of the expected string.

11.3.2 Meta-Grammars in Matrix Representation

Some of the meta-grammars used in Section 8.1 are examined, to see if there is any extra information that we now can extract, given our new measures.

The grammars are:

Grammar 6.6 (Original) To allow the creation of multiple building block structures of different sizes, the mGGAMBB is used as shown in Grammar 6.6 on page 100. When expanding `<bitstring>` there will be a bias towards using building block structures of size >1.

Grammar 8.2 (Equal 1) Here the probability for `<bitstring>` to use a GEGA or building block structures of size >1 is the same, the grammar is shown in Grammar 8.2 on page 130.

Grammar 8.3 (Equal 2) Shown below is a grammar where the probability in `<bitstring>` to use a GEGA or building block structures of any size is the same, see Grammar 8.3 on page 130 and has the same probability for `<bitstring>` to use a GEGA or building block structures of any size.

Grammar 6.1 (GEGA) This is a simple GE approach to GA that does not use a meta-grammar, see Grammar 6.1 on page 96. It is implemented in order to provide a benchmark for the other results. The grammar pre-specifies the number of bit positions in the solution, and the genome is used to select what each bit will become.

Fig. 11.2 shows Grammar 2 with the matrix A as a graph.

Grammar Design Expected Derivation Length

Now the attention is turned to the grammar design of the meta-grammars and what the statistical analysis can reveal. When writing the Q, C, T and calculating the EDL and

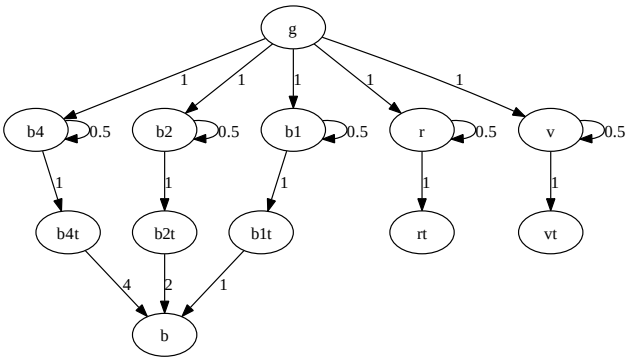
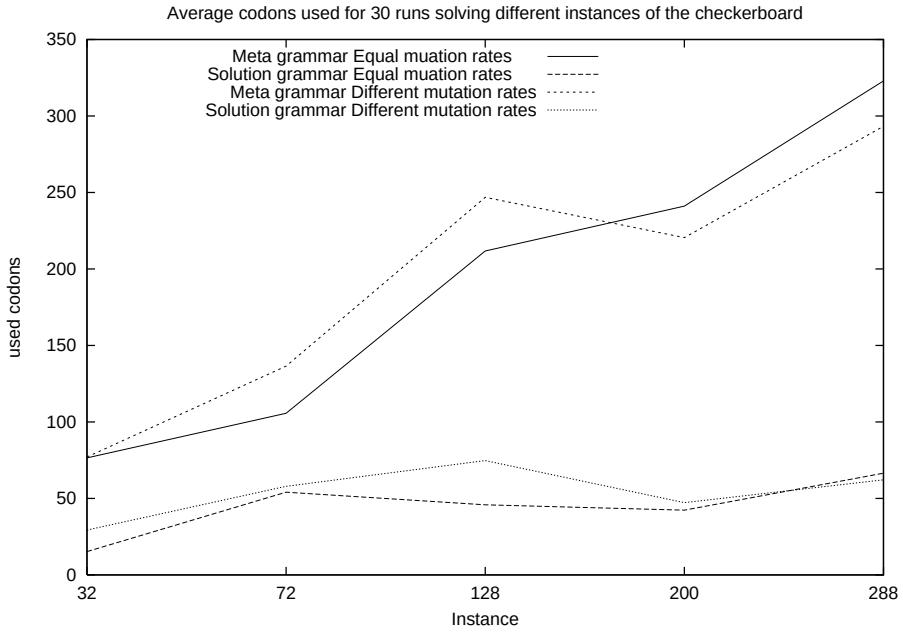


Fig. 11.2: Grammar 2, A

EDW the following results for Grammar 6.6 and 8.2 and 8.3 are obtained.

$$\text{EDL}_{Grammar\ 6.6} = \begin{pmatrix} 35 \\ 12 \\ 8 \\ 6 \\ 5 \\ 3 \\ 2 \\ 4 \\ 1 \\ 1 \\ 4 \\ 1 \end{pmatrix} \quad \text{EDL}_{Grammar\ 8.2} = \begin{pmatrix} 42.77 \\ 12 \\ 8 \\ 6 \\ 5 \\ 3 \\ 2 \\ 11.77 \\ 4.88 \\ 1 \\ 4 \\ 1 \end{pmatrix} \quad \text{EDL}_{Grammar\ 8.3} = \begin{pmatrix} 41 \\ 12 \\ 8 \\ 6 \\ 5 \\ 3 \\ 2 \\ 10 \\ 4 \\ 1 \\ 4 \\ 1 \end{pmatrix}$$

The EDLs for the different grammars show that the expected length for Grammar 8.2 and 8.3 is longer than Grammar 6.6. This is because the Grammar 8.2 and 8.3 add a recursive production in `<rept>`. For Grammar 8.2 `<reps>` is the largest since the recursive probability for Grammar 8.2 is lower. This increase in EDL also affects the EWL similarly



(a) Ave. length (Mutation)

Fig. 11.3: The average length of each codon used at the end of each instance for different mutation rates.

for the grammars used. The relation between the statistical properties of the grammar and the effect of the operators can also be measured.

Expected Derivation Length versus Actual Derivation Length Example

For the original grammar, Grammar 6.6, we study some results to investigate EDL and ADL. In Fig. 11.3 it can be seen that the average number of used codons for Cb_{32} is around twice as large as the EDL (35). This could be an indicator of the over-specification of the grammar. The fact is that it allows many different types of building block structures.

11.4 Discussion

Here we look at how the matrix representation can be used and at implications of its application. First we will show how a matrix representation can calculate values for grammars.

Run Analysis

One follow-up question to the comparisons is, why use the expected derivation length, EDL? Can this measure help improve the mGGA's performance and can it help inform about the grammar representation and the impact on the search? If the grammar is correct for finding the optimal solution the average description length (ADL) should equal the EDL, when enough samples are taken (in a random search through the grammar $\text{ADL}_{n\to\infty} \to \text{EDL}$, n is the number of runs), otherwise the grammar is working against the other operators. A number of cases can be studied (EDL is referred to from the start symbol S, with $|\Omega|$ large enough):

Initialization In the initial population the ADL should equal the EDL if random initialization is used. If ADL < EDL then there are probably too few samples taken.

High fitness With ADL = EDL and the fitness being high for a few generations the grammar will probably be good. If ADL < EDL then the operators are working, but the grammar might not help them. Finally, if ADL > EDL the operators are working, but the number of fitness evaluations might be lowered if the EDL is increased. There might also be some bloat in the solutions.

Low fitness Here ADL = EDL might suggest to change the grammar. If ADL < EDL, change the operators or the grammar since the current region is not successful, and the grammar is not biasing the search enough. With ADL > EDL there might also be some non-beneficial bloat in the population.

Expected Length Expansion and Change Types

The number of unexpanded non-terminals will have an impact on the new size. The size of the expanded rules can be estimated by the EDL. When a change occurs in an individual's production integer input sequence I EDL can be used to bound the size of the

new production integer input I'. This can be done by the observation that

$$|I'| = |I_{0,j}| + \sum_{k \in \delta_{N_j}} EDL(i_k)$$

$$\sum_{k \in \delta_{N_j}} min(D(i_k)) \leq \sum_{k \in \delta_{N_j}} EDL(i_k) \leq \sum_{k \in \delta_{N_j}} max(D(i_k))$$

where the ripple sites, δ_{N_j} is the number of unexpanded non-terminals in the current derivation $\delta_{N_j} = \{x : x \in \delta_j \cap x \in N\}$.

11.5 Summary

A quantitative element to complement the qualitative aspects of a grammar is useful when exploring grammars in GE. This chapter has described properties for grammars in GE, e.g. reachability of sentences. When using a meta-grammar it is useful to be able to decide which solution grammar to explore, and for this there have to be practical grammar measurements. The second part of the chapter described the use of a matrix representation for a PCFG for determining if a derivation would converge, as well as for calculating the expected derivation- and word length. In addition, it is possible to get an overview of the expected symbols in the grammar when altering it and comparing it to other grammars. Finally, an analysis of the fitness and creation of operators using the Expected Derivation Length was discussed.

This ends Part III which has formalized the description of GE and investigated different types of changes to the input and their effect on the output when using a grammar. Furthermore, different approaches to grammar measurement have been studied. Gaps from GE have been covered by the theory and questions raised by the experiments were addressed, e.g. how GE reacts to changes and different operations. In Part IV it is time to reminisce and draw conclusions of our exploration, as well as to propose future work on grammars in GE.

Part IV

Conclusions - Resolution of the Exploration of Grammars in Grammatical Evolution

Chapter 12

Conclusions & Future Work

After having explored grammars in Grammatical Evolution it is time to draw conclusions.

The thesis summary is in Section 12.1. Avenues for future work are discussed in Section 12.2.

12.1 Thesis Summary

This thesis explored grammars in GE with the aim to explore *the role of grammars in Grammatical Evolution*. This can in the long term aid the understanding of how to automatically solve problems. In order to reach this goal the following questions were asked regarding **Performance**

1. *How does the grammar mapping affect the performance of GE?*
2. *Can the use of a meta-grammar improve the performance of GE when problems of a different scale are approached?*
3. *How does the grammar design influence the performance of GE?*

Considering the **Adaptation**

4. *Does the evolutionary learning of a grammar facilitate the capturing of modules?*

Also, regarding **Theory**

5. *How does the representation in GE react to changes in the input?*

6. *How should different grammars be measured and compared?*

In Chapter 4 the grammar mapping was investigated by showing that the order of symbols within a grammar can impact the performance of GE. This was implemented by comparing prefix-, infix- and postfix syntactical variants of a grammar for symbolic regression. The results suggested that the choice of grammar could produce different numbers of invalid solutions for the problems examined, thus affecting performance.

By using meta-grammars it is possible to modify the grammar itself during the evolutionary run. For the problems examined in Chapter 5 we find that irrespective of the representation the presence of automatically defined functions (ADFs) alone is sufficient to improve performance for some problems. In some instances we observe an additional overhead with the adoption of a meta-grammar form of function representation.

The meta-grammars were shown to yield scaling advantages for problems of increasing size in Chapter 6. The exploration of modularity and reuse for an application to GA and the coupling of this into an adaptive representation allow the type and usage of these principles to be evolved through the use of evolvable grammars. For the problem instances examined there are performance advantages for the mGGA when compared to the MGA.

The mGGA was further analyzed by altering the rate of sampling of the evolved solution grammars in meta-grammar GE in Chapter 7. Two approaches were examined, the first adopts implicit sampling using different rates of mutation for the evolved solution grammar versus the solutions sampled from the evolved solution grammar. The second approach explicitly generates more than one sample from each solution grammar in a kind of local-search by randomly mutating the solution chromosome, which is used to construct sentences from the evolved solution grammar. For the problem instances examined, neither approach was found to conclusively improve the performance of the meta-grammar approach to GE in terms of the number of fitness evaluations needed for finding a solution. It is found that the majority of the evolutionary search is currently focused on the generation of the

solution grammars, to such an extent that the candidate solutions are often hard-coded into them, making the solution chromosome effectively redundant.

In Chapter 8 we also set out to measure and understand two more aspects of the mGGA. Firstly, an experiment was undertaken to determine whether a bias in the grammar design used in earlier studies towards the use of building block structures impaired search efficiency. Secondly, we wished to determine if the building block structures were in fact adopted by the population for solving the problem. With respect to the grammar design, we found that this can be an important factor in the search efficiency of the meta-grammar approach for the problems analyzed. A grammar not biased towards building block structures was found to outperform the biased equivalent. An analysis of the adoption of building block structures by the evolutionary search found that these modular structures were used successfully by the population for solving the problem.

A formal description of GE is proposed in Chapter 9 and this provides us with the tools to analyze the impact of changes on the genotype-phenotype mapping of GE. With an improved understanding of how the algorithm works, more efficient search operators can be designed. By focusing the investigation on the grammar representation and distinguishing it from the operators, the essence of GE might be more clearly understood. The behavior of the mGGA in comparison to GE regarding changes in genotype has also been examined, and it has been shown that even changes to terminals on the meta-chromosome mapping are disruptive to the context. This is due to the dependence of the solution on the grammar generated by the meta-chromosome, i.e. all the terminals and non-terminals disrupt as much as the non-terminals. In Chapter 10, the study of changes has revealed a lower bound for the disruption probability in a grammar and thus given some pointers for grammar design. In other words, the fewer non-terminals it contains, the less susceptible it will be to disruption. The effects of a change on the input were labeled. Furthermore a schema theorem for canonical GE has been formulated.

Finally, in Chapter 11 a quantitative element to complement the qualitative aspects of a grammar in GE has been explored. The desirable properties for grammars in GE have been described, e.g. reachability of sentences. Moreover, when using a meta-grammar

it is useful to be able to decide which solution grammar to explore. Both simple and practical measures in EC, e.g. number of rules in the grammar and some more theoretical measures which are more difficult to compute have been described. In addition, a matrix representation for a Probabilistic Context-Free Grammar (PCFG) was introduced in order to determine if a derivation would converge, and we also introduced a matrix representation for the expected derivation length and word length, making it possible to get an overview of the expected symbols in the grammar when altering it and comparing it to other grammars. Finally, an analysis of the fitness and creation of operators using the EDL was discussed.

12.1.1 Contributions

A review of literature in EC regarding algorithms that use grammars, review of the use of grammars in GE and modularity in EC was presented. In addition, the contributions from the thesis are split into three parts, some coming from experiments, others from theory and implementation.

Conclusions from Experiments

Explored grammar mapping Using grammars to examine how the mapping order, i.e. the expansion of non-terminals, is changed it was noted that the number of invalid individuals was tied to the grammar and to the mapping order. Moreover, the results implied that the choice of grammar can produce performance advantage for the problems examined.

Explored meta-grammar & scalability The meta-grammar concept for GE was extended to problems of increasing size. The ability of the meta-grammar to scale to larger problems was confirmed.

Explored meta-grammar & modularity Modularity when using meta-grammars was explored in a fixed-length solution context and a variable-length solution context. The benefits from a modular representation for benchmark problems were verified.

Introduced meta-grammar operators Operators for meta-grammars were introduced and examined. It was confirmed that a slower rate of change for the meta-chromosome can improve performance.

Explored meta-grammar grammars The meta-grammar grammars were explored further, with respect to their bias. The capability of the meta-grammar to use the building block structures provided in the grammar was shown. One recommendation arising from this study is to adopt a meta-grammar that allows the use of both a GA bit string representation in conjunction with the modular building block structures.

Conclusions from Theory

Formally described GE A formal description of GE is proposed and allows us to clearly show the different representations within the algorithm.

Theoretically analyzed the impact of change on GE How an indirect representation from a linear input sequence reacts to changes. Different types of change considering change to input (genotype) were labeled and how these were propagated into other change types in the output (phenotype) via the linear mapping in the CFG. Furthermore, the effects of a change on the input were labeled, as well as a formulation of a schema theorem for canonical GE.

Theoretically analyzed meta-grammar mapping The mapping process involving meta-grammars was explored. The added dependence on the meta-chromosome for the solution chromosome and how the effects of change for a meta-grammar setup behave were also examined.

Introduced a GE schema A GE schema theory was introduced. Some operators were examined in relation to how sequences of the individual genotype are propagated over a generation. It showed that the canonical GE mutation is quite similar to crossover.

Explored grammar measurement in GE We investigated how static analysis can distinguish grammars in EC.

Implementation and Practice

GE software library Developed, implemented and released under an open source license Grammatical Evolution in Java (GEVA) [118], which has been used to implement experiments for several publications.

Grammar measurement in GE A matrix representation for determining the Expected Derivation Length of a grammar was introduced. Furthermore, a binary measurement of the convergence of a PCFG in GE was presented.

Grammar design GE The conclusion is that the fewer non-terminals there are in the grammar, the less susceptible it will be to disruption. The design of the grammar as left or right recursive can delay changes in output from when the input was changed. Reducing deterministic rules in the grammar changes the derivation tree structure, but does not affect the standard GE operators. Grammars which are less affected by the ripple effect can be created by moving non-terminals with fixed subtree size expansions to the begining of the derivation.

Grammar design for mGGA A recommendation arising from the mGGA study is towards the adoption of a meta-grammar that allows the use of both a classic GA bit string representation in conjunction with the modular building block structures.

12.1.2 Limitations of Thesis

This thesis focuses on exploring the grammars in GE. In order to do this, only simple problems have been used to improve the understanding of the grammars. The grammars have been restricted to a sub-part of the CFGs. Moreover, only the grammars in the mapping from genotype to phenotype have been investigated. For example, different mapping orders have not been investigated. The topics of neutrality and locality have also been left out. There are also many good studies regarding schemas in both logic, mathematics, information theory and EC that have not been drawn upon. There is a great potential for

finding additional metrics and analyses for the grammars and schemas which might prove useful. The concept of measures and metrics on any level is basic and not conclusive.

There is a vast literature on formal languages and computational linguistics regarding grammars. There are many different types of grammars. For the Context-Free Grammars only there are myriads of configurations and combinations. Then there are other types of grammars e.g. Attribute Grammars, Type 0 grammars, Tree-Adjoining Grammars etc. We did not consult the Natural Language Processing literature regarding grammar inference and comparison.

When running an evolutionary algorithm, there are a number of direct and indirect choices to be considered, e.g. the choice of operators and parameter values. This thesis has in no way made an exhaustive search of these settings. As for the grammars, neither has there been an exhaustive search of possible grammar combinations here. To allow further generalization the number of problems examined and settings used can always be increased and extended. Moreover, the role of the grammars in association with evolutionary population dynamics has not been investigated.

12.2 Opportunities for Future Research

No exploration is complete without plans for the next expedition. This section suggests subsequent expeditions for GE and grammars. Moreover, after exploration, the focus could also be shifted to exploitation. Thus, the conclusions of this thesis, the scope of its limitations and these limitations themselves could be used as a road map for future work.

Grammar Design

The possibility of generating ADFs [82] dynamically in order to help capture the environment could be investigated further, as could the use of arguments, recursion and higher order functions as in Yu [169] and other implementations of grammar modularity that are not ADFs.

As we have seen there are many ways to write a grammar. One extension of the work

is to use a grammar normal form as a "baseline" grammar that can be used to compare performance. The trick is to find a form which is as unambiguous as possible. Another interesting grammar property is the neighborhood of sentences for a grammar as shown by the grammar mapping order experiments.

Grammar Measurements

The grammar measurements have led to suggestions about adding an objective to the fitness function regarding the diversity of the grammar. More practically, the matrix representation can be used to decide if a grammar will converge or not, which might be helpful for the practitioner when deciding on using a certain grammar for an experiment. There is quite a scope of creating static grammar-analysis software that could be applied to a grammar specified for a GE run. In addition, it could also help visualize grammar bias before a run is initiated.

A tested metric in EC is the fitness distance correlation [18]. The grammar measurements can be incorporated into correlation measurements to see if the grammar and fitness function are working together. Correlations can be useful for adapting the grammar during the run. One example would be string probability fitness correlation.

Meta-Grammar

The meta-grammar GE brings up issues of multi-leveled search for the meta- and the solution grammar. One question here is how this search should be balanced. It could be imagined that evolution would want to get rid of the extra degree of freedom introduced by the meta-grammar with the current operators. Encouraging building block structures will accelerate convergence, thereby reducing diversity. To study operators that maintain diversity and to encourage less hard-coding of the solution grammar would be interesting, since the meta-grammar undoes the work of its intention, decreasing the search space instead of widening it, and this might need to be balanced.

The successful applications of EDAs and EDA-GP could be used as an inspiration

for a more bottom-up implementation of a meta-grammar by creating a new grammar-based algorithm instead of working with the top-down limitation of a large meta-grammar. Therefore, future work could be examining inferring grammars from the solutions generated, solution grammars as well as solutions. The top-down approach of the mGGA model makes the meta-grammar similar to general-to-specific learning, like PEEL, by Shan [143]. The CFG introduces declarative bias, with meta-grammars being one way of introducing more dependencies than the previous codon dependencies.

Operator Design

The balance of exploration and exploitation in GE could be examined by creating an operator that is dependent on the expected change when applied. One improved measure of change in GE would be to take into account all the context changes. Another implementation to reflect a change in the derivation tree is to modify all the codons that correspond to a new context by their $|A_k|$ also (but so that they still would match their derivation). Yet another is to modify the operators so that they normalize the probability for change over the input C. Since the neighborhood of a GE individual defined by single change or by a single application of operators can be the entire search space, it makes sense to describe it as frequency-dependent, as suggested by Altenberg [3].

Further, it could be useful to study operators that perform a more local search [20]. Moreover, context-aware operators for different representation space could be used for identifying linkage in the solution and for finding more linkage-based modules.

The PCFG used in GE could be studied to allow other probabilities than purely uniform, i.e. real values. Such work would need changes to the mapping process, and would also require more specific operators. Another study could be on the impact of not letting deterministic rules use codons, see Eq. (9.7), compared to the use of a codon. The investigation would be regarding the existence of "introns" in the chromosome, as previously done by O'Neill et al. [116].

Application Domains

As such, the empirical results are hard to generalize beyond the problems examined in this thesis. Therefore, it would be interesting to run experiments on SAT, Dynamic Knapsack for mGGA, which has already been tried with a different grammar for GE [25]. The use of pre-, in- and postfix could be tried on a non-continuous problem like the multiplexer [82]. Also, one could try problems which consider modularity as linkage between variables and not just repetitions.

Another interesting approach is that of Langdon and Poli [85] where problems are evolved to learn about particle swarm optimizers and other search algorithms.

Theory

It would be interesting to further study how information theory and compression describe learning, and also to consider what makes a problem 'hard' for GE. Another interesting point about the levels is what distance measures are appropriate for each level. Find measures for changes at all the different GE levels, not only comparing grammars.

Investigate the availability of sequences in the population and the availability of the bias in the search, even if the sequences are not directly preserved there will be a bias from the genetic material available. Here the grammar should affect the search, as discussed in the mod and bucket [74].

The schema theorem could be expanded for more operators in GE. Application of GP concepts such as semantics could hereby aid the search. For GE, tournament selection should be considered to make Eq. (10.13) completely compatible with the implementation in the experiments. Further extensions would be to investigate the probability of the start symbol for the schema to be selected (some grammars are more dependent on the start symbol S). This moves the analysis from schemas more into subtree probabilities.

Some more suggestions: investigate the other representations in GE and classify the changes that can occur due to the mapping. For example, study the changes from the derivation tree to the phenotype.

List of Definitions.

Definition 1 Context-Free Grammar (CFG) . . . 16
Definition 2 Generation . . . 17
Definition 3 Reflexive-Transitive closure . . . 18
Definition 4 Language . . . 18
Definition 5 Probabilistic Context-Free Grammar (PCFG) . . . 19
Definition 6 Derivation . . . 20
Definition 7 Word probability . . . 21
Definition 8 Derivation tree . . . 21
Definition 9 Branch . . . 21
Definition 10 Derivation subtree . . . 21
Definition 11 Derivation difference . . . 21
Definition 12 Individual in GE . . . 147
Definition 13 Codon change . . . 163
Definition 14 Integer production choice change . . . 164
Definition 15 Context change . . . 165
Definition 16 Ripple site . . . 169
Definition 17 Crossover change . . . 171
Definition 18 Redundant change . . . 172
Definition 19 Branch . . . 175
Definition 20 Ripple . . . 176
Definition 21 Schema in CFG . . . 178

List of Examples.

1 CFG . . . 17
2 Sentence generation/derivation . . . 18
3 PCFG . . . 20
4 Word probability . . . 21
5 Boolean grammar . . . 39
6 Choosing a production from a rule . . . 40
7 GE implementation in Python . . . 41
8 GE^2 mapping . . . 78
9 GE individual . . . 148
10 Input of production choices . . . 148
11 Mapping the codon to a production choice . . . 150
12 Delayed changes of production . . . 154
13 Single codon input change . . . 163
14 Input change . . . 165
15 GE derivation change . . . 167
16 CFG designs . . . 169
17 Change effects . . . 174
18 Branch change . . . 175
19 Ripple change . . . 176
20 Ripple Contained . . . 176
21 Reduced grammar . . . 177
22 Derivation string schema . . . 178
23 Grammar complexity . . . 190
24 Rewriting GE grammars . . . 198

Appendix A

Example individuals

Example individuals for Chapter 5.3.7.

```
adf_dyn
Fit:11.0 Phenotype:public Test() {
  super();
  while(getTrail(get_Energy())) {
    adf2();
    if(food_ahead()==1) {
      adf2();
    } else {
      if(food_ahead()==1) {
        adf0();
      } else {
        adf2();
      }
    }
    if(food_ahead()==1) {
      adf0();
    } else {
      if(food_ahead()==1) {
        if(food_ahead()==1) {
          adf1();
        } else {
          adf2();
        }
      } else {
        adf1();
      }
    }
```

```
  }
}
public void adf0() {
  right();
}
public void adf1() {
  if (food_ahead()==1) {
    right();
  } else {
    left();
  }
}
public void adf2() {
  if (food_ahead()==1) {
    move();
  } else {
    left();
  }
  if (food_ahead()==1) {
    right();
  } else {
    if (food_ahead()==1) {
      if (food_ahead()==1) {
        move();
      } else {
        left();
      }
    } else {
      move();
    }
  }
}

adf
Rank:0 Fit:11.0 Phenotype:public Test() {
  super();
  while(getTrail(get_Energy())) {
    if (food_ahead()==1) { right();
    } else { move();
    }
    adf0();
    if (food_ahead()==1) { adf0();
    } else { left();
```

```
    }
    if (food_ahead()==1) { right();
    } else { move();
    }
    adf0();
  }
}
public void adf0() {
  if (food_ahead()==1) { move();
  } else { left();
  }
}

adf_mg
Rank:0 Fit:9.0 Phenotype:Universal:
<prog> ::= public Test() {super(); while(getTrail(get_Energy())) {
    <code> } }  public void adf0() { if (food_ahead()==1) { move(); }
  else { right(); } }
<code> ::= <line> | <code> <line>
<line> ::= <condition> | <op>
<condition> ::= if (food_ahead()==1) { <line> } else { <line> }
<op> ::=  left(); | right(); | move(); | adf<GECodonValue%prog&>();
Solution:public Test() {
  super();
  while(getTrail(get_Energy())) {
    if (food_ahead()==1) {
      left();
    } else {
      move();
    }
    if (food_ahead()==1) {
      if (food_ahead()==1) {
        adf0();
      } else {
        adf0();
    }
    } else {
      right();
    }
    if (food_ahead()==1) {
      if (food_ahead()==1) {
        adf0();
      } else {
```

```
    right();
}
} else {
  if (food_ahead()==1) {
    if (food_ahead()==1) {
      if (food_ahead()==1) {
        move();
      } else {
        left();
    }
    } else {
      if (food_ahead()==1) {
        left();
      } else {
        adf0();
  }
  }
  } else {
    if (food_ahead()==1) {
      right();
    } else {
      adf0();
}
}
}
if (food_ahead()==1) {
  right();
} else {
  if (food_ahead()==1) {
    if (food_ahead()==1) {
      move();
    } else {
      adf0();
  }
  } else {
    if (food_ahead()==1) {
      move();
    } else {
      move();
}
}
}
if (food_ahead()==1) {
```

```
      move();
    } else {
      left();
}
}
}
public void adf0() {
  if (food_ahead()==1) {
    move();
  } else {
    right();
}
}
```

Bibliography

[1] Abdel Latif Abu Dalhoum, Manuel Alfonseca, Manuel Cebrian, Rafael Sanchez-Alfonso, and Alfonso Ortega. Computer-Generated Music Using Grammatical Evolution. In AlAkaidi, M and Geril, P, editor, *MESM '2006: 9TH Middle Eastern Simulation Multiconference*, pages 55–60, 2008. ISBN 978-90-77381-43-4. 9th Middle Eastern Simulation Multiconference, Amman, Jordan, Aug 26-28, 2008.

[2] L. Altenberg. The evolution of evolvability in genetic programming. *Advances in genetic programming*, pages 47–74, 1994.

[3] L. Altenberg. The schema theorem and Price's theorem. *Foundations of genetic algorithms*, 3:23–49, 1995.

[4] Saoirse Amarteifio and Michael O'Neill. Coevolving antibodies with a rich representation of grammatical evolution. In David Corne, Zbigniew Michalewicz, Marco Dorigo, Gusz Eiben, David Fogel, Carlos Fonseca, Garrison Greenwood, Tan Kay Chen, Guenther Raidl, Ali Zalzala, Simon Lucas, Ben Paechter, Jennifier Willies, Juan J. Merelo Guervos, Eugene Eberbach, Bob McKay, Alastair Channon, Ashutosh Tiwari, L. Gwenn Volkert, Dan Ashlock, and Marc Schoenauer, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 904–911, Edinburgh, UK, 2-5 September 2005. IEEE Press. ISBN 0-7803-9363-5.

[5] P.J. Angeline and J.B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 236–241. Lawrence Erlbaum, 1992.

[6] H.J. Antonisse. A grammar-based genetic algorithm. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 193–204. Morgan Kaufmann, 1991.

[7] Thomas Bäck, U. Hammel, and H.P. Schwefel. Evolutionary computation: comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, 1(1):3–17, April 1997. URL `http://ls11-www.cs.uni-dortmund.de/people/schwefel/publications/BHS97.ps.gz`.

[8] W. Banzhaf. Genotype-phenotype-mapping and neutral variation–a case study in genetic programming. *Parallel Problem Solving from Nature–PPSN III*, pages 322–332, 1994.

[9] Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society B*, 57(1):289-300, 1995.

[10] H.G. Beyer and H.P. Schwefel. Evolution strategies–A comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002.

[11] J. Bongard and H. Lipson. Automated reverse engineering of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences of the United States of America*, 2007.

[12] Taylor L. Booth. *Sequential machines and automata theory.* Wiley, 1967.

[13] P.A.N. Bosman and E.D. De Jong. Learning probabilistic tree grammars for genetic programming. *Lecture notes in computer science*, pages 192–201, 2004.

[14] Anthony Brabazon and Michael O'Neill. Credit rating with pi grammatical evolution. In R. Tadeusiewicz, A. Ligeza, and M. Szymkat, editors, *Proceedings of Computer Methods and Systems Conference*, volume 1, pages 253–260, Krakow, Poland, 14-16 November 2005. Oprogramowanie Naukowo-Techniczne Tadeusiewicz. ISBN 83-916420-3-8.

[15] O'Neill M. Brabazon A. *Biologically Inspired Algorithms for Financial Modelling*. Springer, 2006.

[16] M. Brameier and C. Wiuf. Ab initio identification of human microRNAs based on structure motifs. *BMC bioinformatics*, 8(1):478, 2007.

[17] J. Branke, P. Funes, and F. Thiele. Evolving en-route caching strategies for the Internet. *Lecture Notes in Computer Science*, pages 434–446, 2004.

[18] Edmund K. Burke, Steven Gustafson, and Graham Kendall. Diversity in genetic programming: An analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation*, 8(1):47–62, 2004. URL `http://www.cs.nott.ac.uk/~smg/research/publications/gustafson-ieee2004-preprint.pdf`.

[19] J. Byrne, M. O'Neill, E. Hemberg, and A. Brabazon. Analysis of constant creation techniques on the binomial-3 problem in grammatical evolution. In *Evolutionary Computation, 2009. CEC 2009*, Norway, 2009. IEEE Press.

[20] J. Byrne, M. O'Neill, J. McDermott, and A. Brabazon. Structural and nodal mutation in grammatical evolution. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, Montreal, Canada, 2009. ACM.

[21] T. Castle and C.G. Johnson. Positional Effect of Crossover and Mutation in Grammatical Evolution. *LNCS*, 2010.

[22] Manuel Cebrian, Manuel Alfonseca, and Alfonso Ortega. Towards the validation of plagiarism detection tools by means of grammar evolution. *IEEE Transactions on Evolutionary Computation*, 13(3):477–485, June 2009. ISSN 1089-778X. doi: doi:10.1109/TEVC.2008.2008797.

[23] YP Chen, T.L. Yu, K. Sastry, and D.E. Goldberg. A survey of linkage learning techniques in genetic and evolutionary algorithms. *IlliGAL Report*, 2007014, 2007.

[24] N. Chomsky. *Reflections on language*. Pantheon Books, 1975.

[25] Robert Cleary and Michael O'Neill. An attribute grammar decoder for the 01 multiconstrained knapsack problem. In Günther R. Raidl and Jens Gottlieb, editors, *Evolutionary Computation in Combinatorial Optimization – EvoCOP 2005*, volume 3448 of *LNCS*, pages 34–45, Lausanne, Switzerland, 30 March-1 April 2005. Springer Verlag.

[26] O'Neill M. Cleary R. An attribute grammar decode for the 0/1 multiconstrained knapsack problem. In Gottlieb J. Raidl G.R., editor, *European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP), LNCS*, volume LNCS 3448, pages 34–45, Lausanne, Switzerland, 2005. Springer.

[27] T.M. Cover and J.A. Thomas. *Elements of information theory*. Wiley, 2006.

[28] E.D. De Jong and T. Oates. A coevolutionary approach to representation development. In *Proceedings of the icml-2002 workshop on development of representations. Sydney NSW*, volume 2052, 2002.

[29] E.D. De Jong and D. Thierens. Exploiting modularity, hierarchy, and repetition in variable-length problems. *Lecture Notes in Computer Science*, pages 1030–1041, 2004.

[30] E.D. De Jong, D. Thierens, and R.A. Watson. Defining modularity, hierarchy, and repetition. In *GECCO 2004 Workshop Proceedings*, 2004.

[31] I Dempsey. *Grammatical Evolution in Dynamic Environments*. Phd thesis, University College Dublin, 2007.

[32] Ian Dempsey, Michael O'Neill, and Anthony Brabazon. Grammatical constant creation. In Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund Burke, Paul Darwen, Dipankar Dasgupta, Dario Floreano, James Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andy Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Sci-*

ence, pages 447–458, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag. ISBN 3-540-22343-6. URL http://link.springer.de/link/service/series/0558/bibs/3103/31030447.htm.

[33] Ian Dempsey, Michael O'Neill, and Anthony Brabazon. Meta-grammar constant creation with grammatical evolution by grammatical evolution. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1665–1671, Washington DC, USA, 25-29 June 2005. ACM Press. ISBN 1-59593-010-8.

[34] Ian Dempsey, Michael O'Neill, and Anthony Brabazon. *Foundations in Grammatical Evolution for Dynamic Environments*, volume 194 of *Studies in Computational Intelligence*. Springer, April 2009. URL http://www.springer.com/engineering/book/978-3-642-00313-4.

[35] J.A. Draghi, T.L. Parsons, G.P. Wagner, and J.B. Plotkin. Mutational robustness can facilitate adaptation. *Nature*, 463(7279):353–355, 2010.

[36] Damian Eads, Karen Glocer, Simon Perkins, and James Theiler. Grammar-guided feature extraction for time series classification. Technical Report LA-UR-05-4487, Los Alamos National Laboratory, MS D436, Los Alamos, NM, 87545, June 2005.

[37] D. Fagan, M. O'Neill, E. Galván-López, A. Brabazon, and S. McGarraghy. An Analysis of Genotype-Phenotype Maps in Grammatical Evolution. *Genetic Programming*, pages 62–73, 2010.

[38] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence through Simulated Evolution*. Wiley, New York, 1966.

[39] O.O. Garibay. *Analyzing the Effects of Modularity on Search Spaces.* Ph.d. thesis, University of Central Florida Orlando,Florida, 2008.

[40] Ozlem O. Garibay, Ivan I. Garibay, and Annie S. Wu. The modular genetic algorithm: Exploiting regularities in the problem space. In Adnan Yazici and Cevat Sener, editors, *Computer and Information Sciences - ISCIS 2003, 18th International Symposium, Antalya, Turkey, November 3-5, 2003, Proceedings*, volume 2869 of *Lecture Notes in Computer Science*, pages 584–591. Springer, 2003. ISBN 3-540-20409-1.

[41] D. Gavrilis and I.G. Tsoulos. Classification of fetal heart rate using grammatical evolution. *Signal Processing Systems Design and Implementation, 2005. IEEE Workshop on*, pages 425–429, 2005. ISSN 1520-6130.

[42] L. Georgiou and W.J. Teahan. Experiments with Grammatical Evolution in Java. *Computational Intelligence (SCI)*, 102:45–62, 2008.

[43] John S. Gero, Sushil J. Louis, and Sourav Kundu. Evolutionary learning of novel grammars for design improvement. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 8:83–94, 1994.

[44] A.L. Gibbs and F.E. Su. On choosing and bounding probability metrics. *International Statistical Review/Revue Internationale de Statistique*, 70(3):419–435, 2002.

[45] David E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms.* Kluwer Academic Publishers, Norwell,MA,USA, 2002.

[46] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Professional, January 1989. ISBN 0201157675.

[47] D.E. Goldberg, K. Deb, H. Kargupta, and G. Harik. Rapid, accurate optimization of difficult problems using fast messy genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, 1993.

[48] Steven Gustafson. http://www.gustafsonresearch.com/research/vis/, 2008.

[49] T. Håndstad, A.J.H. Hestnes, and P. Sætrom. Motif kernel generated by genetic programming improves remote homology and fold detection. *BMC bioinformatics*, 8 (1):23, 2007.

[50] R. Harper and A. Blair. A structure preserving crossover in grammatical evolution. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 3, 2005.

[51] Robin Harper and Alan Blair. A self-selecting crossover operator. In Gary G. Yen, Lipo Wang, Piero Bonissone, and Simon M. Lucas, editors, *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 5569–5576, Vancouver, 6-21 July 2006. IEEE Press. ISBN 0-7803-9487-9.

[52] Robin Harper and Alan Blair. Dynamically defined functions in grammatical evolution. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 9188–9188, Vancouver, 6-21 July 2006. IEEE Press. ISBN 0-7803-9487-9.

[53] MA Harrison. *Introduction to formal language theory*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1978.

[54] Y. Hasegawa and H. Iba. Latent variable model for estimation of distribution algorithm based on a probabilistic context-free grammar. *IEEE Transactions on Evolutionary Computation*, 13(4):858–878, 2009.

[55] Yoshihiko Hasegawa and Hitoshi Iba. Estimation of distribution algorithm based on probabilistic grammar with latent annotations. In Dipti Srinivasan and Lipo Wang, editors, *2007 IEEE Congress on Evolutionary Computation*, page 1043?1050, Singapore, 25-28 September 2007. IEEE Computational Intelligence Society, IEEE Press.

[56] Yoshihiko Hasegawa and Hitoshi Iba. Estimation of bayesian network for program generation. In The Long Pham, Hai Khoi Le, and Xuan Hoai Nguyen, editors, *Proceedings of the Third Asian-Pacific workshop on Genetic Programming*, pages 35–46, Military Technical Academy, Hanoi, VietNam, 2006.

[57] E. Hemberg. An exploration of learning and grammars in grammatical evolution. In *Proceedings of the 11th annual conference companion on Genetic and evolutionary computation conference*, pages 2705–2708. ACM, 2009.

[58] E. Hemberg, N. McPhee, M. O'Neill, and A. Brabazon. Pre-,in-and postfix grammars for symbolic regression in grammatical evolution. Derry, 2008.

[59] E. Hemberg, M. O'Neill, and A. Brabazon. Altering Search Rates of the Meta and Solution Grammars in the mGGA. *Lecture Notes in Computer Science*, 4971:362, 2008.

[60] Erik Hemberg, Conor Gilligan, Michael O'Neill, and Anthony Brabazon. A grammatical genetic programming approach to modularity in genetic algorithms. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 1–11, Valencia, Spain, 11 - 13 April 2007. Springer. ISBN 3-540-71602-5.

[61] Erik Hemberg, Michael O'Neill, and Anthony Brabazon. Grammatical bias and building blocks in meta-grammar grammatical evolution. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 3775–3782, 2008.

[62] Erik Hemberg, Michael O'Neill, and Anthony Brabazon. An investigation into automatically defined function representations in grammatical evolution. In *Mendel 2009 15th International Conference on Soft Computing Brno*, pages 67–73, June 24-26 2009.

[63] J.F. Hicklin. Application of the genetic algorithm to automatic program generation. Master's thesis, University of Idaho, 1986.

[64] N.X. Hoai and RI McKay. A framework for tree adjunct grammar guided genetic

programming. In *Proceedings of the Post-graduate ADFA Conference on Computer Science (PACCS01)*, pages 93–99. Citeseer, 2001.

[65] N.X. Hoai, RI McKay, and D. Essam. Representation and structural difficulty in genetic programming. *IEEE Transactions on evolutionary computation*, 10(2):157, 2006.

[66] Gregory S. Hornby. Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1729–1736, Washington DC, USA, 25-29 June 2005. ACM Press. ISBN 1-59593-010-8. URL `http://www.cs.bham.ac.uk/~wbl/biblio/gecco2005/docs/p1729.pdf`.

[67] J. Hugosson, E. Hemberg, A. Brabazon, and M. O'Neill. Genotype Representations in Grammatical Evolution. *Applied Soft Computing*, 10(1):36–43, 2009.

[68] CM Johnson and S. Feyock. A genetics-based technique for the automated acquisition of expertsystem rule bases. In *Developing and Managing Expert System Programs, 1991., Proceedings of the IEEE/ACM International Conference on*, pages 78–82, 1991.

[69] H. Kargupta and S. Ghosh. Toward machine learning through genetic code-like transformations. *Genetic Programming and Evolvable Machines*, 3(3):231–258, 2002.

[70] Nadav Kashtan, Elad Noor, and Uri Alon. Varying environments can speed up evolution. *PNAS*, pages 0611630104+, August 2007. doi: 10.1073/pnas.0611630104. URL `http://dx.doi.org/10.1073\%2Fpnas.0611630104`.

[71] M. Keijzer, C. Ryan, M. O'Neill, and M. Cattolico. Ripple crossover in genetic programming. *LNCS 2038*, pages 74–86, 2001.

[72] M. Keijzer, C. Ryan, M. O'Neill, M. Cattolico, and V. Babovic. Adaptive logic programming. In *Genetic and Evolutionary Computation Conference GECCO 2001*, pages 42–49, San Francisco, CA, USA, 2001. Morgan Kaufmann.

[73] Maarten Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 70–82, Essex, 14-16 April 2003. Springer-Verlag. ISBN 3-540-00971-X.

[74] Maarten Keijzer, Michael O'Neill, Conor Ryan, and Mike Cattolico. Grammatical evolution rules: The mod and the bucket rule. *LNCS*, 2278:123–130, 2002. URL `http://link.springer.de/link/service/series/0558/papers/2278/22780123.pdf`.

[75] R.E. Keller and W. Banzhaf. Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. In *Proceedings of the First Annual Conference on Genetic Programming*, pages 116–122. MIT Press, 1996.

[76] R.E. Keller and R. Poli. Cost-benefit investigation of a genetic-programming hyperheuristic. In *Proceedings of the Evolution artificielle, 8th international conference on Artificial evolution*, pages 13–24. Springer-Verlag, 2007.

[77] Robert E. Keller and Wolfgang Banzhaf. The evolution of genetic code in genetic programming. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proc. of the Genetic and Evolutionary Computation Conf. GECCO-99*, page 1077?1082, San Francisco,CA, 1999. Morgan Kaufmann.

[78] Robert E. Keller and Wolfgang Banzhaf. The evolution of genetic code in genetic programming. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proc. of the Genetic and Evolutionary Computation Conf. GECCO-99*, pages 1077–1082, San Francisco, CA, 1999. Morgan Kaufmann.

[79] M. Kim, C.W. Ahn, M. Médard, and M. Effros. On minimizing network coding resources: An evolutionary approach. In *Proc. NetCod*. Citeseer, 2006.

[80] Donald E. Knuth. Backus normal form vs. backus naur form. *Commun. ACM*, 7(12): 735–736, 1964. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/355588.365140.

[81] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994. ISBN 0-262-11189-6.

[82] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press, December 1992. ISBN 0262111705.

[83] K. Krawiec and B. Wieloch. Functional modularity for genetic programming. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 995–1002. ACM, 2009.

[84] W. B. Langdon and A. P. Harrison. Evolving regular expressions for genechip probe performance prediction. In G Rudolph, T Jansen, S Lucas, C Poloni, and N Beume, editors, *Parallel Problem Solving From Nature - PPSN X, Proceedings*, volume 5199 of *Lecture Notes In Computer Science*, pages 1061–1070, 2008. ISBN 978-3-540-87699-1. 10th International Conference on Parallel Problem Solving from Nature, Dortmund, Germany, SEP 13-17, 2008.

[85] W. B. Langdon and Riccardo Poli. Evolving problems to learn about particle swarm optimisers and other search algorithms. *IEEE Transactions on Evolutionary Computation*, 11(5):561–578, October 2007. ISSN 1089-778X.

[86] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002. ISBN 3-540-42451-2. URL `http://www.cs.ucl.ac.uk/staff/W.Langdon/FOGP/`.

[87] E. Lehman and A. Shelat. Approximation algorithms for grammar-based compression. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 205–212. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 2002.

[88] Hod Lipson, Jordan B. Pollack, and Nam P. Suh. Promoting modularity in evolutionary design. In *Proceedings of DETC'01 2001 ASME Design Engineering Technical Conferences September 9-12, 2001, D ETC2001 IDAC-21099*, Pittsburgh, Pennsylvania, USA, November 08 2001. URL `http://citeseer.ist.psu.edu/521058.html;http://www.mae.cornell.edu/lipson/papers/promod.pdf`.

[89] David M. W. Luerssen, Martin H. Powers. Evolvability and redundancy in shared grammar evolution. In *IEEE Congress on Evolutionary Computation 2007.*, pages 370–377, September 2007.

[90] Martin Luerssen and David Powers. Evolving encapsulated programs as shared grammars. *Genetic Programming and Evolvable Machines*, 9(3):203–228, September 2008. URL `http://dx.doi.org/10.1007/s10710-008-9061-2`.

[91] Steve Margetts and Antonia J. Jones. An adaptive mapping for developmental genetic programming. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 97–107, Lake Como,Italy, 18-20 April 2001. Springer-Verlag.

[92] T. McConaghy and G. Gielen. Canonical form functions as a simple means for genetic programming to evolve human-interpretable functions. In *Proceedings of the*

8th annual conference on Genetic and evolutionary computation, page 862. ACM, 2006.

[93] R. I. (Bob) McKay, X. H. Nguyen, P. A. Whigham, and Y. Shan. Grammars in genetic programming: A brief review. In L. Kang, Z. Cai, and Y. Yan, editors, *Progress in Intelligence Computation and Intelligence: Proceedings of the International Symposium on Intelligence, Computation and Applications*, pages 3–18, Wuhan, PRC, April 2005. China University of Geosciences Press. ISBN 7-5625-1983-8. URL `http://sc.snu.ac.kr/PAPERS/isica05.pdf`.

[94] RI McKay, J. Shin, T.H. Hoang, X.H. Nguyen, and N. Mori. Using compression to understand the distribution of building blocks in genetic programming populations. In *IEEE Congress on Evolutionary Computation, 2007. CEC 2007*, pages 2501–2508, 2007.

[95] BA McKinney and D. Tian. Grammatical immune system evolution for reverse engineering nonlinear dynamic bayesian models. *Cancer Informatics*, 6:433, 2008.

[96] N.F. McPhee, E. Crane, S.E. Lahr, and R. Poli. Developmental plasticity in linear genetic programming. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1019–1026. ACM, 2009.

[97] JM Mingo and R. Aler. Grammatical evolution guided by reinforcement. In *IEEE Congress on Evolutionary Computation, 2007. CEC 2007*, pages 1475–1482, 2007.

[98] David J. Montana. Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA, 7 May 1993. URL `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/stgp.ps.Z`.

[99] Marco A. Montes de Oca. Exposing a bias toward short-length numbers in grammatical evolution. In M. O'Neill, L. Vanneschi, A. I. Esparcia-Alczar S. Gustafson, I. De Falco, A. Della Cioppa, and E. Tarantino, editors, *LNCS 4971. Proceedings of*

the Eleventh European Conference on Genetic Programming. EuroGP 2008, pages 278–288, Berlin, 2008. Springer.

[100] Alison Motsinger-Reif, Theresa Fanelli, Anna Davis, and Marylyn Ritchie. Power of grammatical evolution neural networks to detect gene-gene interactions in the presence of error. *BMC Research Notes*, 1(1):65, 2008. ISSN 1756-0500. doi: 10.1186/1756-0500-1-65. URL `http://www.biomedcentral.com/1756-0500/1/65`.

[101] Eoin Murphy, Michael O'Neill, Edgar Galvan, and Anthony Brabazon. Tree-adjunct grammatical evolution. In *2010 IEEE World Congress on Computational Intelligence*. IEEE Press, 2010.

[102] JE Murphy, H. Carr, and M. O'Neill. Grammatical Evolution for Gait Retargeting. In *Proc. Theory and Practice of Computer Graphics*, pages 159–162, 2008.

[103] Miguel Nicolau. Automatic grammar complexity reduction in grammatical evolution. In R. Poli, S. Cagnoni, M. Keijzer, E. Costa, F. Pereira, G. Raidl, S. C. Upton, D. Goldberg, H. Lipson, E. de Jong, J. Koza, H. Suzuki, H. Sawai, I. Parmee, M. Pelikan, K. Sastry, D. Thierens, W. Stolzmann, P. L. Lanzi, S. W. Wilson, M. O'Neill, C. Ryan, T. Yu, J. F. Miller, I. Garibay, G. Holifield, A. S. Wu, T. Riopka, M. M. Meysenburg, A. W. Wright, N. Richter, J. H. Moore, M. D. Ritchie, L. Davis, R. Roy, and M. Jakiela, editors, *GECCO 2004 Workshop Proceedings*, Seattle, Washington, USA, 26-30 June 2004.

[104] M. O Neill, A. Brabazon, M. Nicolau, S. Mc Garraghy, and P. Keenan. pigrammatical evolution. *Lecture Notes In Computer Science*, pages 617–629, 2004.

[105] A.R. Oganov and C.W. Glass. Evolutionary crystal structure prediction as a tool in materials design. *Journal of Physics: Condensed Matter*, 20:064210, 2008.

[106] M. O'Neill, C. Ryan, M. Keijzer, and M. Cattolico. Crossover in Grammatical Evolution. *Genetic Programming and Evolvable Machines*, 4(1):67–93, 2003.

[107] M. O'Neill, F. Leahy, and A. Brabazon. Grammatical swarm: A variable-length particle swarm algorithm. *Swarm Intelligent Systems*, pages 59–74, 2006.

[108] M. O'Neill, J.M. Swafford, J. McDermott, J. Byrne, A. Brabazon, E. Shotton, C. McNally, and M. Hemberg. Shape grammars and grammatical evolution for evolutionary design. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1035–1042. ACM, 2009.

[109] M. O'Neill, J. McDermott, J.M. Swafford, J. Byrne, E. Hemberg, E. Shotton, C. McNally, A. Brabazon, and M. Hemberg. Evolutionary design using grammatical evolution and shape grammars: Designing a shelter. *International Journal of Design Engineering*, 2010. Forthcomming.

[110] Michael O'Neill and Anthony Brabazon. Grammatical differential evolution. In Hamid R. Arabnia, editor, *IC-AI*, pages 231–236. CSREA Press, 2006. ISBN 1-932415-96-3.

[111] Michael O'Neill and Anthony Brabazon. mGGA: The meta-grammar genetic algorithm. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 311–320, Lausanne, Switzerland, 30 March - 1 April 2005. Springer. ISBN 3-540-25436-6.

[112] Michael O'Neill and Conor Ryan. Automatic generation of caching algorithms. In Kaisa Miettinen, Marko M. Mäkelä, Pekka Neittaanmäki, and Jacques Periaux, editors, *Evolutionary Algorithms in Engineering and Computer Science*, pages 127–134, Jyväskylä, Finland, 30 May - 3 June 1999. John Wiley & Sons. URL `http://www.mit.jyu.fi/eurogen99/papers/oneill.ps`.

[113] Michael O'Neill and Conor Ryan. Grammar based function definition in grammatical evolution. In Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors, *Proceedings of the Genetic and Evolutionary*

Computation Conference (GECCO-2000), pages 485–490, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann. ISBN 1-55860-708-0. URL `http://www.cs.bham.ac.uk/~wbl/biblio/gecco2000/GP226.pdf`.

[114] Michael O'Neill and Conor Ryan. Grammatical evolution by grammatical evolution: The evolution of grammar and genetic code. In Maarten Keijzer, Una-May O'Reilly, Simon M. Lucas, Ernesto Costa, and Terence Soule, editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 138–149, Coimbra, Portugal, 5-7 April 2004. Springer-Verlag. ISBN 3-540-21346-5.

[115] Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, Norwell, MA, USA, 2003. ISBN 1402074441.

[116] Michael O'Neill, Conor Ryan, and Miguel Nicolau. Grammar defined introns: An investigation into grammars, introns, and bias in grammatical evolution. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 97–103, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann. ISBN 1-55860-774-9. URL `http://www.cs.bham.ac.uk/~wbl/biblio/gecco2001/d01.pdf`.

[117] Michael O'Neill, Robert Cleary, and Nikola Nikolov. Solving knapsack problems with attribute grammars. In R. Poli, S. Cagnoni, M. Keijzer, E. Costa, F. Pereira, G. Raidl, S. C. Upton, D. Goldberg, H. Lipson, E. de Jong, J. Koza, H. Suzuki, H. Sawai, I. Parmee, M. Pelikan, K. Sastry, D. Thierens, W. Stolzmann, P. L. Lanzi, S. W. Wilson, M. O'Neill, C. Ryan, T. Yu, J. F. Miller, I. Garibay, G. Holifield, A. S. Wu, T. Riopka, M. M. Meysenburg, A. W. Wright, N. Richter, J. H. Moore, M. D. Ritchie, L. Davis, R. Roy, and M. Jakiela, editors, *GECCO 2004 Workshop*

Proceedings, Seattle, Washington, USA, 26-30 June 2004. URL `http://www.cs.bham.ac.uk/~wbl/biblio/gecco2004/WGEW003.pdf`.

[118] Michael O'Neill, Erik Hemberg, Conor Gilligan, Eliott Bartley, James McDermott, and Anthony Brabazon. Geva:grammatical evolution in java. *SIGEVOlution*, 3(2): 17–23, Summer 2008.

[119] Hemberg E. O'Neill M., Brabazon A. Subtree deactivation control with grammatical genetic programming. In *IEEE Congress on Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence)*, pages 3768–3774, Hong Kong, 2008. IEEE Press.

[120] Ryan C. O'Neill M. *Grammatical Evolution. Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, 2003.

[121] U.M. O'Reilly. Investigating the generality of automatically defined functions. In *Proceedings of the first annual conference on genetic programming*, pages 351–356. MIT Press, 1996.

[122] Una-May O'Reilly and Martin Hemberg. Integrating generative growth and evolutionary computation for form exploration. *Genetic Programming and Evolvable Machines*, 8(2):163–186, 2007. ISSN 1389-2576. doi: http://dx.doi.org/10.1007/s10710-007-9025-y.

[123] A. Ortega, M. de la Cruz, and M. Alfonseca. Christiansen grammar evolution: Grammatical evolution with semantics. *Evolutionary Computation, IEEE Transactions on*, 11(1):77–90, Feb. 2007. ISSN 1089-778X. doi: 10.1109/TEVC.2006.880327.

[124] J. Parent, A. Nowe, K. Steenhaut, and A. Defaweux. Linear Genetic Programming using a compressed genotype representation. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 2, 2005.

[125] N. Paterson and M. Livesey. Evolving caching algorithms in C by genetic programming. *Genetic Programming*, pages 262–267, 1997.

[126] Norman R. Paterson and Mike Livesey. Distinguishing genotype and phenotype in genetic programming. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, pages 141–150, Stanford University, CA, USA, 28–31 July 1996. Stanford Bookstore. ISBN 0-18-201031-7. URL `ftp://ftp.dcs.st-and.ac.uk/pub/norman/GADS.ps.gz`.

[127] Wojciech Piaseczny, Hideaki Suzuki, and Hidefumi Sawai. Chemical genetic programming – evolutionary optimization of the genotype-to-phenotype translation set. *Artificial Life and Robotics*, 9(4):202–208, 2005.

[128] R. Poli and C.R. Stephens. The building block basis for genetic programming and variable-length genetic algorithms. *International Journal of Computational Intelligence Research*, 1(1-2):183–197, 2005.

[129] Riccardo Poli and Nicholas Freitag McPhee. A linear estimation-of-distribution gp system. In *EuroGP*, pages 206–217, 2008.

[130] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. URL `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

[131] Alain Ratle and Michele Sebag. Grammar-guided genetic programming and dimensional consistency: application to non-parametric identification in mechanics. *Applied Soft Computing*, 1(1):105–118, June 2001.

[132] J. Reddin, J. McDermott, and M. O'Neill. Elevated Pitch: Automated Grammatical Evolution of Short Compositions. *Proceedings of the EvoWorkshops 2009 on Applications of Evolutionary Computing: EvoCOMNET, EvoENVIRONMENT, EvoFIN, EvoGAMES, EvoHOT, EvoIASP, EvoINTERACTION, EvoMUSART, EvoNUM, EvoSTOC, EvoTRANSLOG*, pages 579–584, 2009.

[133] Evandro Nunes Regolin and Aurora Trindad Ramirez Pozo. Bayesian automatic programming. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 38–49, Lausanne, Switzerland, 30 March - 1 April 2005. Springer. ISBN 3-540-25436-6.

[134] Philipp Rohlfshagen, Per Kristian Lehre, and Xin Yao. Dynamic evolutionary optimisation: an analysis of frequency and magnitude of change. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1713–1720, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-325-9. doi: http://doi.acm.org/10.1145/1569901.1570131.

[135] B.J. Ross. The evaluation of a stochastic regular motif language for protein sequences. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 120–128, 2001.

[136] Brian J. Ross. Logic-based genetic programming with definite clause translation grammars. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1236, Orlando,Florida,USA, 13-17 1999. Morgan Kaufmann. URL `citeseer.ist.psu.edu/article/ross99logicbased.html`.

[137] Franz Rothlauf and David E. Goldberg. *Representations for Genetic and Evolutionary Algorithms*. Physica-Verlag, 2002. ISBN 3790814962.

[138] Franz Rothlauf and Marie Oetzel. On the locality of grammatical evolution. In Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anik Ekrt, editors, *EuroGP*, volume 3905 of *Lecture Notes in Computer Science*, pages 320–330. Springer, 2006.

[139] G. Rudolph. Finite Markov chain results in evolutionary computation: a tour d'horizon. *Fundamenta Informaticae*, 35(1-4):67–89, 1998.

[140] C. Ryan, A. Azad, A. Sheahan, and M. O Neill. No Coercion and No Prohibition, A Position Independent Encoding Scheme for Evolutionary Algorithms-The Chorus System. *Lecture notes in computer science*, pages 131–141, 2002.

[141] C. Ryan, M. Nicolau, and M. O Neill. Genetic algorithms using grammatical evolution. *Lecture notes in computer science*, pages 278–287, 2002.

[142] C. Ryan, M. Keijzer, and M. Nicolau. On the avoidance of fruitless wraps in grammatical evolution. *Lecture Notes in Computer Science*, pages 1752–1763, 2003.

[143] Yin Shan. *Program Distribution Estimation with Grammar Models*. PhD thesis, University of New South Wales, 2005.

[144] Yin Shan, Robert McKay, Daryl Essam, and Hussein Abbass. A survey of probabilistic model building genetic programming. ALAR Technical Report Series TR-ALAR-200510014, The Artificial Life and Adaptive Robotics Laboratory, School of Information Technology and Electrical Engineering University of New South Wales Northcott Drive,Campbell,Canberra,ACT 2600 Australia, 2005. URL `http://www.itee.adfa.edu.au/~alar/techreps/2005100014.pdf`.

[145] J. Shin, M. Kang, R.I. McKay, X. Nguyen, T. Hoang, N. Mori, and D. Essam. Analysing the regularity of genomes using compression and expression simplification. *Lecture Notes in Computer Science*, 4445:251, 2007.

[146] Herbert A. Simon. *The sciences of the artificial (3rd ed.)*. MIT Press, Cambridge,MA,USA, 1996.

[147] L. Spector. Simultaneous evolution of programs and their control structures. *Advances in genetic programming*, 2:137–154, 1996.

[148] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, 2002.

[149] H. Suzuki, H. Sawai, and W. Piaseczny. Chemical genetic algorithms-evolutionary optimization of binary-to-real-value translation in genetic algorithms. *Artificial Life*, 12(1):89–115, 2006.

[150] I. Tanev. Genetic programming incorporating biased mutation for evolution and adaptation of Snakebot. *Genetic Programming and Evolvable Machines*, 8(1):39–59, 2007.

[151] M. Toussaint. *The evolution of genetic representations and modular neural adaptation.* PhD thesis, Ruhr-Universität Bochum, 2003.

[152] I.G. Tsoulos, D. Gavrilis, and E. Dermatas. GDF: A tool for function estimation through grammatical evolution. *Computer physics communications*, 174(7):555–559, 2006.

[153] S. Ventura, C. Romero, A. Zafra, J.A. Delgado, and C. Hervás. JCLEC: a Java framework for evolutionary computation. *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, 12(4):381–392, 2008.

[154] G.P. Wagner. Adaptation and the modular design of organisms. In *Advances in artificial life: Third European Conference on Artificial Life, Granada, Spain, June 4-6, 1995: proceedings*, page 317. Springer Verlag, 1995.

[155] G.P. Wagner and L. Altenberg. Complex adaptations and the evolution of evolvability. *Evolution*, 50(3):967–976, 1996.

[156] R.A. Watson. *Compositional evolution.* PhD thesis, Brandeis University, 2002.

[157] CS Wetherell. Probabilistic languages: A review and some open questions. *ACM Computing Surveys (CSUR)*, 12(4):361–379, 1980.

[158] P. A. Whigham. Grammatically-based genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City,California,USA, 9 1995. URL `citeseer.ist.psu.edu/whigham95grammaticallybased.html`.

[159] P. A. Whigham. A schema theorem for context-free grammars. In *In 1995 IEEE Conference on Evolutionary Computation*, pages 178–181. IEEE Press, 1995.

[160] P.A. Whigham. Inductive bias and genetic programming. *Genetic Algorithms in Engineering Systems: Innovations and Applications, 1995. GALESIA. First International Conference on (Conf. Publ. No. 414)*, pages 461–466, Sep 1995.

[161] Peter Alexander Whigham. *Grammatical Bias for Evolutionary Learning*. PhD thesis, School of Computer Science,University College,University of New South Wales,Australian Defence Force Academy, Canberra,Australia, 14 October 1996. URL `http://divcom.otago.ac.nz/sirc/Peterw/Publications/thesis.zip`.

[162] Bill C. White, Joshua C. Gilbert, David M. Reif, and Jason H. Moore. A statistical comparison of grammatical evolution strategies in the domain of human genetics. In David Corne, Zbigniew Michalewicz, Marco Dorigo, Gusz Eiben, David Fogel, Carlos Fonseca, Garrison Greenwood, Tan Kay Chen, Guenther Raidl, Ali Zalzala, Simon Lucas, Ben Paechter, Jennifier Willies, Juan J. Merelo Guervos, Eugene Eberbach, Bob McKay, Alastair Channon, Ashutosh Tiwari, L. Gwenn Volkert, Dan Ashlock, and Marc Schoenauer, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 491–497, Edinburgh, UK, 2-5 September 2005. IEEE Press. ISBN 0-7803-9363-5.

[163] D. Wilson and D. Kaur. Search, Neutral Evolution, and Mapping in Evolutionary Computing: A Case Study of Grammatical Evolution. *IEEE Transactions on Evolutionary Computation*, 13(3):566–590, 2009.

[164] Garnett Wilson and Malcolm Heywood. Introducing probabilistic adaptive map-

ping developmental genetic programming with redundant mappings. *Genetic Programming and Evolvable Machines*, 8(2):187–220, 2007. ISSN 1389-2576. doi: http://dx.doi.org/10.1007/s10710-007-9027-9.

[165] G.C Wilson. *Probabilistic Adaptive Mapping Developmental Genetic Programming*. PhD thesis, Dalhousie University, Halifax, Nova Scotia, March 2007.

[166] GC Wilson, A. Mc Intyre, and MI Heywood. Resource review: Three open source systems for evolving programs–lilgp, ECJ and grammatical evolution. *Genetic Programming and Evolvable Machines*, 5(1):103–105, 2004.

[167] Man Leung Wong. Evolving recursive programs by using adaptive grammar based genetic programming. *Genetic Programming and Evolvable Machines*, 6(4):421?455, 2005. doi: http://dx.doi.org/10.1007/s10710-005-4805-8.

[168] J. Woodward. Modularity in Genetic Programming. In *Genetic Programming, Proceedings of EuroGP 2003*, pages 14–16. Springer, 2003.

[169] T. Yu. Hierarchical processing for evolving recursive and modular programs using higher-order functions and lambda abstraction. *Genetic Programming and Evolvable Machines*, 2(4):345–380, 2001.

[170] T.L. Yu. *A matrix approach for finding extrema: problems with modularity, hierarchy, and overlap*. PhD thesis, University of Illinois at Urbana-Champaign Champaign, IL, USA, 2006.