DOCTOR OF PHILOSOPHY

Constituent grammatical evolution

Georgiou, Loukas

Award date:
2012

Awarding institution:
Bangor University

CONSTITUENT GRAMMATICAL EVOLUTION

By

LOUKAS GEORGIOU

BSc, University of Wales, Bangor
2004

A Dissertation
Submitted in Partial Fulfilment of the Requirements for the degree of Doctor of Philosophy

Supervised by Dr. William J. Teahan

External Examiner:
Dr. Michael O'Neill

College of Physical & Applied Sciences
School of Computer Science

BANGOR UNIVERSITY
Bangor, Wales

2012

Abstract

Evolutionary algorithms are a competent nature-inspired approach for complex computational problem solving. One recent development is Grammatical Evolution, a grammar-based evolutionary algorithm which uses genotypes of variable length binary strings and a unique genotype-to-phenotype mapping process based on a BNF grammar definition describing the output language that is able to create valid individuals of an arbitrary structure or programming language.

This study surveys Grammatical Evolution, identifies its most important issues, investigates the competence of the algorithm in a series of agent-oriented benchmark problems, provides experimental results which cast doubt about its effectiveness and efficiency on problems involving the evolution of the behaviour of an agent, and presents Constituent Grammatical Evolution (CGE), a new innovative evolutionary automatic programming algorithm. CGE extends Grammatical Evolution by incorporating the concepts of constituent genes and conditional behaviour-switching. It builds from elementary and more complex building blocks a control program which dictates the behaviour of an agent and it is applicable to the class of problems where the subject of search is the behaviour of an agent in a given environment. Experimental results show that the new algorithm significantly improves Grammatical Evolution in all problems it has been benchmarked.

Additionally, the investigation undertaken in this work required the development of a series of tools which are presented and described in detail. These tools provide an extendable open source and publicly available framework for experimentation in the area of evolutionary algorithms and their application in agent-oriented environments and complex systems.

Preface

It was November of 2005 when – during the evolutionary computation literature review I was conducting as part of this study – my supervisor Dr. William J. Teahan introduced me to a new evolutionary algorithm, named Grammatical Evolution. While he was in Poland for a conference, he met up two researchers from Ireland – Michael O'Neill and Anthony Brabazon. Dr. William J. Teahan was convinced that Grammatical Evolution (pioneered by Ryan, Collins and Michael O'Neill) was a promising approach meriting attention and further research. He pointed out the ability of this new algorithm to produce arbitrary valid data structures and executable programs. Furthermore, he found it very interesting that the programs being evolved could be constrained through the use of a BNF grammar definition and that domain knowledge could be more easily incorporated by choosing a particular grammar.

Grammatical Evolution really appealed to me right from the beginning. There was already a strong research activity about this new algorithm and a series of publications, providing insights to the algorithm, its issues, its variations, and future research directions. All these helped me to understand this new approach and to appreciate its strong advantages over other evolutionary algorithms as well as some of its weaknesses and disadvantages. It was at that point where the basis for this thesis was formed which finally resulted in the work presented here.

I hope that the reader enjoys this text as much I enjoyed the journey from that November until today, and that this work will contribute to the field by providing useful information and insights and stimulating further work and ideas to researchers interested in the application of evolutionary algorithms for the emergence of behaviours of agents.

Acknowledgements

I would like to state how grateful I am to my wife Ero for the patience she showed during this work. She is the person who prompted me to start this endeavour and who supported me all these years both morally and indeed by taking care of providing me the necessary time for concentrating on this research. Ero is also the inspirer and creator of the art work (logos and graphics) of the presentations, posters, and web sites that resulted from this work.

I would like to apologise to my two children, Dimitra and Harris, for the time I was not with them. Both were born during this research and I had very little time with them until now. Fortunately, this journey is coming to the end and I will be able to give to them and to my wife the time and attention they really deserve.

I would like also to thank my parents for inspiring in me the thirst for knowledge and for making me the person who I am today. Unfortunately, they are no longer in life but I am sure their thoughts were with me all these years.

Finally, I owe a great debt of gratitude to my supervisor Dr. William J. Teahan for his insightful comments and suggestions as well as his guidance and support on this work. I would like to thank him for his timely response to any question or issue I had and for being always on my side every time I needed him. Also, he is the person who guided me to investigate the area of evolutionary computation and in particular of Grammatical Evolution. I'm really grateful for this. I hope that this thesis will be not the end, but just the beginning of a new and more fascinating journey – the quest of understanding the nature of human knowledge and intelligence.

Declaration and Consent

Details of the Work

I hereby agree to deposit the following item in the digital repository maintained by Bangor University and/or in any other repository authorized for use by Bangor University.

Author Name: Loukas Georgiou

Title: Constituent Grammatical Evolution

Supervisor/Department: Dr. William J. Teahan / School of Computer Science

Funding body (if any): ---

Qualification/Degree obtained: PhD

This item is a product of my own research endeavours and is covered by the agreement below in which the item is referred to as "the Work". It is identical in content to that deposited in the Library, subject to point 4 below.

Non-exclusive Rights

Rights granted to the digital repository through this agreement are entirely non-exclusive. I am free to publish the Work in its present version or future versions elsewhere.

I agree that Bangor University may electronically store, copy or translate the Work to any approved medium or format for the purpose of future preservation and accessibility. Bangor University is not under any obligation to reproduce or display the Work in the same formats or resolutions in which it was originally deposited.

Bangor University Digital Repository

I understand that work deposited in the digital repository will be accessible to a wide variety of people and institutions, including automated agents and search engines via the World Wide Web.

I understand that once the Work is deposited, the item and its metadata may be incorporated into public access catalogues or services, national databases of electronic theses and dissertations such as the British Library's EThOS or any service provided by the National Library of Wales.

I understand that the Work may be made available via the National Library of Wales Online Electronic Theses Service under the declared terms and conditions of use (http://www.llgc.org.uk/index.php?id=4676). I agree that as part of this service the National Library of Wales may electronically store, copy or convert the Work to any approved medium or format for the purpose of future preservation and accessibility. The National Library of Wales is not under any obligation to reproduce or display the Work in the same formats or resolutions in which it was originally deposited.

Contents

List of Figures

List of Listings

List of Equations

List of Tables

List of the Accompanying Material

The accompanying CD includes the following material:

1. **Dissertation.**

   Electronic copies of the dissertation, in a structure and format that is exactly the same as the printed version.

   a. Dissertation in MS Word format.
   b. Dissertation in PDF format.

2. **Published Papers.**

   a. jGE – A Java implementation of Grammatical Evolution, (2006).
   b. Implication of Prior Knowledge and Population Thinking in Grammatical Evolution: Toward a Knowledge Sharing Architecture, (2006).
   c. Experiments with Grammatical Evolution in Java, (2008).
   d. Grammatical Evolution and the Santa Fe Trail Problem, (2010).
   e. Constituent Grammatical Evolution, (2011).

3. **The jGE Library.**

   a. Binary and Source code.
   b. Documentation.
   c. Quick Start Guide and BNF sample files.

4. **The jGE NetLogo extension.**

   a. Binary and Source code.
   b. Documentation.

5. **NetLogo Models.**

   a. Santa Fe Trail Simulation.
   b. GE and CGE Model for the Santa Fe Trail problem.
   c. Los Altos Hills Simulation.
   d. GE and CGE Model for the Los Altos Hills problem.
   e. Hampton Court Maze Simulation.
   f. GE and CGE Model for the Hampton Court Maze problem.
   g. Chevening House Maze Simulation.
   h. GE and CGE Model for the Chevening House Maze problem.

6. **Experiments Log files**

Chapter 1

Introduction

1.1 Background and Motivation

Evolutionary Computation is a subfield of Artificial Intelligence, more particularly of Computational Intelligence. It takes inspiration from natural evolution and genetics for the creation of population-based meta-heuristic search and optimisation algorithms intended to solve complex computational problems. Evolutionary algorithms have been applied with success in a variety of fields and problems of a static or dynamic nature, such as function optimisation, machine learning and self-adaptation to name a few. Even though important drawbacks and issues have been identified – the lack of a formal model that unifies them, the computational cost of these algorithms, and the lack of an effective modelling of the meta-learning based progress of biological evolution – their potential makes them an attractive choice for application in areas that resemble features of real-world problems like coping with the evolution and adaptation of emergent agents behaviours.

One of the most important developments in evolutionary computation is Genetic Programming (Koza, 1992) which directly evolves computer programs. Grammatical Evolution, first published by Ryan, Collins and O'Neill (1998), is a form of Genetic Programming that uses variable length binary genomes which govern how a Backus Naur Form (BNF) grammar definition is mapped to an executable computer program written in an arbitrary programming language. Due to an innovative mapping formula, a genome wrapping mechanism, and the use of variable length genomes, this new approach provides a solution to the "closure" problem of Genetic Programming and grammar-based automatic programming algorithms, namely the issue of generation and preservation of valid programs (O'Neill and Ryan, 2001). According to the Grammatical Evolution literature, this algorithm outperforms Genetic Programming in problems such as symbolic regression and the Santa Fe Trail which is an instance of the artificial ant problem (O'Neill and Ryan, 2001), and has proved to be successful in a series of static and

dynamic real world problems (O'Neill and Ryan, 2003; Brabazon and O'Neill, 2006; Dempsey, O'Neill and Brabazon, 2009).

But the success of Grammatical Evolution on the Santa Fe Trail has come into question with Robilliard, et al. (2006) claiming that the comparison with Genetic Programming in this problem, as it is conducted in the GE literature, is not a fair one. Furthermore, research on Grammatical Evolution has revealed important issues with the most noticeable of them being destructive crossovers (O'Neill, Ryan, Keijzer and Cattolico, 2003), the genome bloating (Harper and Blair, 2006b), the dependency problems (Ryan, Collins and O'Neill, 1998), and the low locality of genotype-to-phenotype mapping (Rothlauf and Oetzel, 2006).

Besides the identified issues, Grammatical Evolution seems to be a promising approach that deserves further research. The use of a BNF grammar definition enables the evolution of arbitrary structures in a very flexible way. The encoding of the genome as variable length binary strings simplifies the genetic operations such as crossover and mutation. The resolution of the "closure" problem allows the creation of valid individuals. These are some of the most important advantages of Grammatical Evolution making it an appealing approach for natural-inspired solving of complex problems.

Consequently, one of the first questions that arose prior to the research detailed in this thesis was whether the claim of Robilliard, et al. (2006) was true. If yes, what is the real performance of Grammatical Evolution in the Santa Fe Trail? Is the method still outperforming Genetic Programming in this problem? Also, is it possible to achieve a performance improvement of this algorithm by tackling some of its reported issues, and how much? These questions formed the motivation for the research undertaken in this thesis which aims to provide answers and support them with experimental results.

1.2 Aims and Objectives

The aims and objectives of the research undertaken in this thesis are as follows:

- to survey the literature of the area of evolutionary computation – with a focus on Grammatical Evolution – and point out research findings, open issues, and future directions;

- to identify important Grammatical Evolution issues and possible resolutions under research;
- to investigate the claim of Robilliard, et al. (2006) that when Grammatical Evolution literature benchmarks this grammar-based algorithm against Genetic Programming in the Santa Fe Trail problem, a biased search space is used which is not semantically equivalent with the original as defined by Koza (1992);
- if this claim is proved to be true, to benchmark the performance of Grammatical Evolution in the Santa Fe Trail problem when the original search space is used, and answer the question whether it still outperforms Genetic Programming; and
- to tackle some of the most noticeable issues of Grammatical Evolution with the hope to improve its effectiveness and efficiency in benchmark problems where the subject of evolution is the behaviour of an agent in a given environment.

When the research of this thesis was started, there was no publicly available implementation of Grammatical Evolution in the Java programming language that would enable benchmarking and experimentation with this algorithm in a platform independent manner. Therefore, an implementation had to be developed. The resultant toolkit, named jGE Library, is also intended to provide a framework for the addition of more evolutionary algorithms, not only Grammatical Evolution, and for the development of experimental implementations.

The benchmarking of Grammatical Evolution and its proposed improvement requires also the development of simulations of the problems in question and of the evolutionary runs to be conducted. For this reason there is the need for the implementation of a series of models and simulations in a multi-agent programmable modelling environment such as NetLogo. To this end, an extension of jGE for the NetLogo modelling environment, named jGE NetLogo extension, has been developed and it is hoped that this will allow both communities of NetLogo and Grammatical Evolution to utilise the features and advantages of each other.

1.3 Thesis Statement

Grammatical Evolution is a flexible and promising approach to evolutionary computation, due to its unique features and advantages, which merits further investigation for

improvement besides the reported issues of this algorithm which cast doubt about its effectiveness and efficiency. The performance of Grammatical Evolution can be increased significantly – at least in problems where the subject is the emergence of the behaviour of an agent – by confronting the issues of: semantic changes in the original search space which exclude good solutions of the problem in question; destructive crossovers; and genotype bloating.

The proposed improvement of the algorithm, named Constituent Grammatical Evolution, aims to reduce the impact of the above issues with the incorporation of the concepts of conditional behaviour-switching and constituent genes, and with imposing a limit to the genotype size. The achieved increase in effectiveness and efficiency is demonstrated with a series of experiments in benchmark problems which involve the evolution of the behaviour of simple agents in static environments. The experiments are facilitated with the development of a general-purpose toolkit and the appropriate models for the simulation of these benchmarks.

1.4 Thesis Contributions

Previous to this study, there had been no other detailed work investigating and confirming the claim of Robilliard, et al. (2006) that Grammatical Evolution literature uses a biased search space when its performance is compared against Genetic Programming in the Santa Fe Trail problem. It is proved through a series of different experiments that the imposed bias is very large and that good solutions of the problem are excluded. Furthermore it is shown that Grammatical Evolution does not outperform Genetic Programming on the Santa Fe Trail when it uses the original search space as defined by Koza (1992) and when a configuration similar to this mentioned in the GE literature (O'Neill and Ryan, 2001; 2003, p.56) is applied.

Also, this is the first reported publication of the following: the benchmarking of Grammatical Evolution to highlight the impact of using the original and a GE literature SFT-like biased search space, on Los Altos Hills, a more difficult version of the artificial ant problem than the Santa Fe Trail; and on two particular maze searching problems, Hampton Court Maze and Chevening House Maze. It is shown that Grammatical Evolution cannot solve the Los Altos Hills problem and that it does not perform well on

Hampton Court Maze and Chevening House Maze, regardless of which of these two search spaces is used. These results put in question the competence of Grammatical Evolution on agent-oriented problems.

This thesis proposes an improvement of Grammatical Evolution, named Constituent Grammatical Evolution, which copes with how to bias the search space toward useful areas without excluding good candidate solutions, how to reduce the impact of destructive crossover events, and how to resolve the genotype bloating issue. This new algorithm achieves a higher success rate and finds better solutions than Grammatical Evolution in the Santa Fe Trail problem. It is also shown that the new algorithm improves Grammatical Evolution in three more benchmarks: Los Altos Hills, Hampton Court Maze, and Chevening House Maze.

Additionally, this work demonstrates the effects of the application of declarative grammatical bias, through the addition of building blocks or the encoding of knowledge, on the performance of Grammatical Evolution in the Santa Fe Trail problem and provides experimental evidence which highlights the importance of the number of the added building blocks. Furthermore, it demonstrates the impact of genotype bloat to the computational effort of the genotype-to-phenotype mapping process during a Grammatical Evolution run and the significant decrease that can be achieved, without affecting the performance in terms on finding a solution, when a genotype size limit above a problem specific threshold is enforced.

Finally, for the investigation undertaken in this work, the development of a set of tools was required. One of them, the jGE Library, was the first published implementation of Grammatical Evolution in Java (Georgiou and Teahan, 2006a) and is freely available under the GNU general public licence from its web site (Georgiou, 2006). Another tool is the jGE NetLogo extension which is the only implementation of Grammatical Evolution for the NetLogo modelling environment. Finally, the development of a series of NetLogo models was required that simulate the benchmark problems and perform the evolutionary runs.

1.5 Thesis Summary

Chapter 2 provides a background to the field of evolutionary computation, with a focus on Grammatical Evolution, and on how evolutionary algorithms are evaluated. The chapter begins by pointing out the nature-inspired influences to evolutionary computation. Then it provides a historical review of the field and discusses its current issues, findings, and future directions. The chapter continues with the description of Grammatical Evolution and its molecular biology influences. Then, the chapter presents a study of Grammatical Evolution standard configuration, reviews the past and current research on this new evolutionary algorithm, lists its applications and variations, and discusses its main issues and future research opportunities. After reviewing research on modularity in Genetic Programming and Grammatical Evolution and the most important freely available implementations of Grammatical Evolution, in a variety of programming languages, the chapter closes with the presentation of standard benchmarking problems that were used for the evaluation of evolutionary algorithms, mainly in the area of Genetic Programming.

Chapter 3 presents the jGE Library, an implementation of Grammatical Evolution in the Java programming language, which has been developed to provide a framework for the implementation and experimentation with evolutionary algorithms and to facilitate the experiments discussed in later chapters. This chapter explains the architecture and the main components of this toolkit and demonstrates how it can be used and configured through examples, code samples and proof-of-concept experiments in solving Hamming distance, symbolic regression and trigonometric identity problems. Also, some Java issues which were revealed during the development of the library are discussed. The chapter closes with a comparison of jGE with two widely known Grammatical Evolution implementations, libGE and GEVA.

Chapter 4 presents some extensions to the jGE Library. The chapter discusses how jGE can be extended with the incorporation of natural inspired concepts, such as prior knowledge and population thinking, and discusses the experimental results. Finally, the chapter presents the jGE Library extension for the NetLogo modelling environment, and shows through examples how it can be used in NetLogo models.

The next chapters of the thesis deal with three of the Grammatical Evolution issues (the changed definition of the Santa Fe Trail benchmark which additionally excludes good solutions, the destructive crossovers events, and the genotype bloating) and their proposed confrontation.

Chapter 5 discusses the Grammatical Evolution literature issue of the usage of a search space that is semantically different to the original in the Santa Fe Trail benchmark when it is compared to Genetic Programming. Namely, the application in the GE grammar of a declarative language bias which additionally excludes good solutions of the problem. Then the chapter describes the NetLogo models that were developed to facilitate the investigation of the issue through experimentation, and provides the results in a series of experiments conducted with jGE and the NetLogo models. The chapter closes with a discussion of the experimental results and the conclusions.

Chapter 6 investigates the effects of grammatical bias in the performance of Grammatical Evolution in the Santa Fe Trail problem and whether a bias can be enforced that increases at the same time both the effectiveness (success rate) and efficiency (solution quality) without changing the semantics of the search space as defined in the original problem by Koza (namely, with using grammars stating the same language bias). The chapter opens with a presentation of previous work on grammatical bias, the issues that arise when language or search bias are used, and how modularity and knowledge encoding may bias the search for a solution. Then, a series of experiments using a variety of biased grammars in the Santa Fe Trail problem is presented, and the chapter closes with a discussion of the experimental results and the conclusions.

Chapter 7 presents Constituent Grammatical Evolution, a variation of Grammatical Evolution that aims to reduce the risk of excluding good solutions when a biased grammar is used as well as to reduce the impact of destructive crossovers and to confront the genotype bloating issue. The chapter analyses the main concepts of this new algorithm, provides a detailed description of it, and shows in detail how it is implemented in a standard Genetic Programming benchmark, the artificial ant problem. Then the chapter compares it against Grammatical Evolution in four benchmark problems and discusses the results. Before concluding, the chapter presents a series of more experiments and statistical results that further investigate aspects of the proposed Grammatical Evolution variation.

Chapter 8 summarises the work undertaken and presented in this thesis, discusses the findings and provides the conclusions. Then the chapter reviews the aims and objectives, and summarises the contributions of the thesis before the suggested future work is identified.

Appendix A contains a list of print and on-line publications which have arisen from this work; Appendix B overviews the jGE web site; Appendix C provides a description of the Constituent Grammatical Evolution web site; Appendix D lists the people who have shown interest in the jGE Library; Appendix E provides guide lines on how to use jGE; Appendix F gives detailed class diagrams of the main components of the jGE Library; and Appendix G provides a description of the main procedures of the jGE NetLogo extension.

Chapter 2

Background and Literature Review

2.1 Evolutionary Computation

2.1.1 Introduction

Ernst Mayr (2002, p.8) notes that everything on Earth seems to be in a continuous flux and mentions three types of changes: regular changes, irregular changes, and evolution. A regular change is for example the change from day to night and back again. The movement of the tectonic plates is an example of irregular change. Evolution however, is a particular kind of change that seems to keep going continuously and to have a directional component.

Evolution as a theory was first developed in detail with the 1859 publication of the book *On the Origin of Species*, by Charles Darwin and it is the most important concept in biology (Mayr 2002, p.xiii). Charles Darwin's theory of evolution through natural selection is based on population thinking and states that "a population or species changes through the continuous production of new genetic variation and through the elimination of most members of each generation, because they are less successful either in the process of the non-random elimination of individuals or in the process of sexual selection i.e., they have less reproductive success" (Mayr 2002, p.83). Indeed, Charles Darwin in his seminal work *On the Origin of Species* says that "It is not the strongest of the species that survive, nor the most intelligent, but the one most responsive to change" (Ghanea-Hercock 2003, p.119).

According to Evolutionary Synthesis (Mayr 2002, p.9), evolution is change in the properties of populations of organisms over time. In other words, the population is the so-called unit of evolution. Genes, individuals, and species also play a role, but it is the change in populations that characterises organic evolution. Seven of the processes which are responsible for the genetic turnover in a population are: 1) Selection, 2) Mutation, 3)

Gene Flow, 4) Genetic Drift, 5) Biased Variation, 6) Movable Elements, and 7) Non-random mating (Mayr 2002, pp.103-111).

The importance of the concept of evolution goes far beyond biology and alongside with the developments in genetics by Gregor Mendel and the discovery of the structure of the DNA by James Watson and Francis Crick, inspired, amongst other things, the field of evolutionary computation (Ghanea-Hercock 2003, pp.19-26).

Hirsh (2000) notes that evolutionary computation is based on the idea that basic concepts of biological reproduction and evolution can serve as a metaphor on which computer-based, goal-directed problem solving can be based. The general idea is that a computer program can maintain a population of artefacts represented using some suitable computer-based data structures. Elements of that population can then mate, mutate, or otherwise reproduce and evolve, directed by a fitness measure that assesses the quality of the population with respect to the goal of the task at hand.

De Jong (2006, p.1) defines evolutionary computation as the use of evolutionary systems as computational processes for solving complex problems.

According to Ghanea-Hercock (2003, p.iii), the breakthrough in understanding how to apply evolution in computational systems is normally credited to the group led by John Holland and their seminal work on the Genetic Algorithm, in 1975 at the University of Michigan.

2.1.2 History and Main Approaches

De Jong (2006, p.1) traces the roots of the evolutionary computation field as far back as the 1930s but as he notes it was the 1960s, when inexpensive digital computing technology emerged, where the whole field progressed. De Jong, identifies three prominent groups of scientists and engineers who saw the computer simulation of evolutionary systems as a means to better understand complex evolutionary systems: Evolutionary Biologists (development and testing of models of natural evolutionary systems), Computer Scientists and Engineers (harnessing the power of evolution to build useful new artefacts), and Artificial Life Researchers (design and experimentation with new and interesting artificial evolutionary worlds).

The history and foundation of evolutionary computation can be summarised in the following milestones:

- Friedberg (1958), Evolutionary Algorithm approach for automatic programming;
- Rechenberg (1965), Evolution Strategies;
- Fogel, Owens and Walsh (1966), Evolutionary Programming;
- John Holland (1967), Genetic Algorithm;
- John Koza (1992), Genetic Programming.

According to Ghanea-Hercock (2003, p.2), the first evolutionary algorithm in the form most commonly utilised today was the Genetic Algorithm (GA) created by John Holland and his students at the University of Michigan, and the next major development in Evolutionary Algorithm (EA) systems was Genetic Programming, introduced by John Koza in 1992.

De Jong (2006, pp.23-31) provides a comprehensive historical perspective of the field of evolutionary computation. He notes that in the 1930s, the influential ideas of Sewell Wright (1932) lead quite naturally to the notion of an evolutionary system as an optimisation process (even today optimisation problems are the most dominant application area of evolutionary computation). Friedman (1956) viewed an evolutionary system as a complex, adaptive system that changes its makeup and its responses over time as it interacts with a dynamically changing landscape. Friedberg's work (1958) is an early example of automating the process of writing computer programs based on evolutionary processes. Rechenberg (1965) used evolutionary processes to solve difficult real-valued parameter optimisation problems. From these ideas emerged the family of algorithms called "Evolution Strategies". Fogel, Owens and Walsh (1966) saw the potential of achieving the goals of artificial intelligence via evolutionary techniques. Intelligent agents were presented as finite state machines and an evolutionary framework called "Evolutionary Programming" evolved better finite state machines (agents) over time. Holland saw evolutionary processes as a key element in the design and implementation of robust adaptive systems that were capable of dealing with an uncertain and changing environment (Holland, 1962; 1967). His work lead to an initial family of "reproductive plans" which formed the basis for what we call "Simple or Generational Genetic Algorithms" today.

De Jong (2006, pp.23-31) notes that the last three paradigms mentioned above (Evolutionary Programming, Evolution Strategies, and Generational Genetic Algorithms) have been unified (late 1980s and early 1990s) to the field known as "Evolutionary Computation" today. Since the 1990s, new evolutionary algorithms emerged in this new field such as Genitor, Genetic Programming, Messy Genetic Algorithms, Samuel, CHC, Genocoop, and other. He argues that today, the field of Evolutionary Computation has the look and feel of a mature scientific discipline and there are various groups of scientists and engineers who focus on a number of fundamental extensions to existing Evolutionary Algorithm models in order to improve and extend their problem solving capabilities.

Furthermore, David Fogel (2006, pp.59-87) reviews some of the first attempts and contributions to the formal application of the evolutionary theory principles to practical engineering problems. He notes that such attempts appeared first in the areas of statistical process control, machine learning, and function optimisation. Namely, Evolutionary Operation or EVOP (Box, 1957), A Learning Machine: Herman (Friedberg, 1958), Artificial Life (Barricelli, 1957), Evolutionary Programming (Fogel, Owens and Walsh, 1966), Evolution Strategies (Rechenberg, 1965), and Genetic Algorithms (Holland, 1975).

According to Ghanea-Hercock (2003, pp.47-56), the latest significant development in the area of evolutionary computation is Genetic Programming (Cramer, 1985; Koza, 1992; 1994; Koza, et al., 1999; 2003; Langdon and Poli, 2001) named and popularised by John Koza (1992). This variation of Holland's Genetic Algorithms has the main characteristic that it performs its search not with fixed-length binary encoded strings but in variable-length tree-based representations of complete computer programs. Indeed, it introduces a new significant concept: The modular subroutine approach termed Automatically Defined Functions (ADFs).

Also, O'Neill and Ryan (2003, pp.5-21) conduct a comprehensive survey of Evolutionary Automatic Programming (EAP) systems, which are defined as those systems that adopt evolutionary computation to automatically generate computer programs (Genetic Programming, Binary GP, AIM GP, developmental GP), and they mention some of the diverse approaches that have been adopted in the field of EAP: a) Tree-based systems (Genetic Programming), b) Grammar-based GP systems (review in McKay, et al., 2010) (cellular encoding of neural networks, bias in GP, Genetic Programming Kernel, combination of GP and Inductive Logic Programming, Auto-Parallelisation of GP), and c)

String-based GP (Binary GP, Machine Code GP or AIM-GP, Genetic Algorithm for Deriving Software or GADS, CFG/GP).

The latest developments in evolutionary computation show a significant popularity of Parallel Evolutionary Algorithms, which is to be expected because evolution is an inherently parallel process (Fogel 2006, p.87). Alba and Tomassini (2002) and Alba (2005, pp.108-112) provide a unified and comprehensive survey of the area of parallel evolutionary algorithms (PEAs). They classify the evolutionary algorithms to the following broad classes: Panmictic Evolutionary Algorithms (Generational and Steady State) and Structured Evolutionary Algorithms (Distributed EA - dEA, Cellular EA - cEA, and Non-Standard Structured EAs e.g. injection island GA, gradual distributed real-code GA, coevolutionary approaches, and more).

Cantú-Paz (2001, pp.6-11) suggests the following categorization of the techniques used to parallelise GAs: 1) single-population master-slave GAs, 2) multiple-population GAs, 3) fine-grained GAs, and 4) hierarchical hybrids. He notes in Cantú-Paz (1998) that parallel GAs are capable of combining speed and efficacy and he argues that the only way to achieve a greater understanding of parallel GAs is to study individual facets independently noting that the most influential publications in parallel GAs concentrate on only one aspect (migration rates, communication topology, or deme size) either ignoring or making simplifying assumptions on the others.

Indeed, research has been conducted on implementations of genetic algorithms on parallel hardware using standard parallel approach (master – slave) and decomposition approach (Adamidis, 1994). The models of the last approach are: the coarse-grained parallel algorithm / migration model (island model, stepping-stone model, and other coarse-grained models) and the fine-grained parallel genetic algorithm / diffusion or isolation-by-distance or neighbourhood model.

Today, some of the most common applications of evolutionary algorithms according to De Jong (2006, pp.71-113) are a) Optimisation (Parameter optimisation, constrained optimisation, data structure optimisation, multi-objective optimisation), b) Searching, c) Machine Learning, d) Automated Programming, and e) Adaptation of agent behaviour.

2.1.3 Current Issues

Baum (2004, pp.124-125) argues that the problem with evolutionary algorithms using sexual recombination is that they usually abandon credit assignment. Namely, when half the genotype is changed due to crossover, the fitness is largely randomised. This kind of change does not slowly adjust segments of the genotype so that they work well in the context of all other segments, and consequently this approach abandons credit assignment. Instead, biological evolution makes progress using a meta-learning process. It discovers semantically meaningful genes and can swap them around in a way so that these changes correspond to real modular structure in the world. Baum claims that evolutionary algorithms have not yet achieved an effective modelling of this progress.

Ghanea-Hercock (2003, pp.101-114), claims that three of the fundamental problems of the Evolutionary Algorithms which have to be addressed are: a) the computational cost in terms of processing power and memory requirements, b) the optimal selection of the operators and their parameters, and c) the complex nature of the evolutionary algorithms search space. Indeed, the mapping of concepts of Darwinian Evolution onto computers results in the creation of analogously complex and diverse computational artefacts (Hirsh, 2000).

De Jong (2006, pp.71-113) focuses on computer science and engineering issues relating to how evolutionary algorithms can be used to solve difficult computational problems, using at the same time the biological and dynamical system perspectives as a source of ideas for better and more effective algorithms. He argues that a solid conceptual starting point for EA-based problem solving is to conceive of a simple EA as a heuristic search procedure that uses objective fitness feedback to explore complex solution spaces effectively. He notes that in order for this procedure to be effective for a specific class of problems, some key design decisions have to be made involving: a) Deciding what an individual in the population represents; b) providing a means of computing the fitness of an individual; c) deciding how children (new search points) are generated from parents (current search points); d) specifying population sizes and dynamics; e) defining a termination criterion for stopping the evolutionary process; and f) returning an answer.

Dempsey, O'Neill and Brabazon (2009, p.30) mention the following main issues of evolutionary algorithms when they are applied in dynamic environments:

- EAs have a tendency to eventually converge to an optimum and thereby lose their diversity necessary for efficiently exploring the search space and consequently their ability to adapt to a change in the environment when such change occurs.
- Researchers often over fit (their) algorithms to various classes of static optimisation problems with the focus on getting the representation and operators to produce rapid convergence to near optimal points.

Dempsey, O'Neill and Brabazon (2009, p.30) state that populations in GAs exhibit a tendency to converge to an optimum with a resulting loss in diversity. This impedes the algorithm from efficiently exploring new areas in the solution space when the optimum shifts. Five approaches researchers have adopted in overcoming this issue are the following (Dempsey, O'Neill and Brabazon 2009, pp.30-31):

1. equipping algorithms with a memory mechanism (explicit or implicit) to recall effective solutions found in previous generations;
2. preserving diversity in the population in order to prevent the noted problem of convergence in a population;
3. achieving a balance between exploration and exploitation by assigning sub-populations to specific areas of the search space;
4. breaking the problem to its fundamental pieces (problem decomposition); and
5. achieving evolvability by providing a representation that aids the production of fitter offspring than their parents.

De Jong (2006, pp.23-31) argues that some of the issues being continuously explored are: a) the positive and negative aspects of various phenotype and genotype representations; b) the inclusion of Lamarckian properties to allow the inheritance of acquired traits; c) non-random parents selection for mating and speciation of a population allowing mating only within species; d) decentralised, highly parallel models which exploit the natural parallelism of evolutionary algorithms; e) self-adapting systems which adapt the EA parameters during an EA run; f) cooperative and competitive co-evolutionary systems, inspired by the symbiotic relationships and the arms race met in nature between populations, respectively; and g) agent-oriented models with individuals reacting with their environment.

Finally, O'Neill, Vanneschi, Gustafson and Banzhaf (2010) outline some of the challenges and open issues in the field of Genetic Programming and in some cases suggest ways to overcome them. They discuss the following issues: identifying appropriate representations; identifying how hard a particular problem will be for some GP systems; how GP performs in dynamic problems; how much detail is necessary to adopt from the biological paradigm of natural evolution; design of an evolutionary system capable of continuously adapting and searching; generalisation; benchmarks; scalability and modularity; complexity; the halting problem; how much domain knowledge should be injected in GP algorithms; GP usefulness for Biology; constants; GP theory; distributed models; and usability of GP.

2.1.4 Research Findings and Future Directions

According to Fogel (2006, pp.167-170), the theoretical and empirical analysis of evolutionary computation found that the most fundamental property of an optimisation algorithm is its convergence in the limit. Obtaining practical convergence rate information is difficult and this has resulted in past years in exhaustive empirical research on specific problems. But the no free lunch theorem of Wolpert and Macready (1997) proves that improved performance of any algorithm indicates a match between the structure of the algorithm and the structure of the problem. Therefore, as Fogel claims, the empirical analysis alone is not reliable. It must be accompanied and supported by relevant theory, and as he notes, many of the early efforts of explaining how evolutionary algorithms work were flawed.

Indeed, Fogel (2006, pp.170-171) argues that studies and research during the last decade showed that the *k*-armed bandit analysis as it was offered in the mid-1970s was not well formed, proportional selection does not generate an "optimal allocation of trials," binary representations are not optimal for function optimisation, no specific cardinality of representation is generally superior, evolutionary algorithms are not uniformly improved when recombination is employed, the schema theorem (Holland, 1975) may not apply when fitness is a random variable, and self-adaptation does not require recombination to be successful.

Fogel (2006, p.171) claims that since there is no best evolutionary algorithm (single choice of mutation, recombination, selection, population size, etc.), the Fitness

Distribution offers a tool for the assessing and trading-off of the different parameter choices in specific circumstances. But "determining the appropriate design choices by classes of function, initialisation, representation, and so forth, remains an open question – one that cannot be answered by examining any of these facets in isolation."

De Jong (2006, p.69), unifies with his work the canonical evolutionary algorithms (Evolutionary Programming, Evolution Strategies, and Genetic Algorithms) identifying the following basic elements of an evolutionary algorithm:

- A parent population of size m.
- An offspring population of size n.
- A parent selection method.
- A survival selection method.
- A set of reproductive operators (mutation and recombination).
- An internal method of representing individuals.

De Jong (2006, p.69) states that adopting this unified view and understanding the role each of these elements is playing, leads to the understanding of the limitations of the canonical algorithms and encourages the design of new and more effective algorithms.

Also, he argues (De Jong 2006, p.49) that experimental and theoretical work that has been undertaken until now makes clear that there are no universally optimal methods (there is not a unique configuration of the above basic elements which effectively solves any class of problems). Indeed, he states (De Jong 2006, pp.66-67) that achieving a proper balance between the elements that increase variation (exploration) and those that decrease variation (exploitation) is one of the most important factors in designing evolutionary algorithms. The exploration's level can be controlled by the reproductive mechanisms (mutation, recombination) and the exploitation's level can be controlled by the selection mechanisms (parents selection for reproduction and offspring selection for survival).

Ghanea-Hercock (2003, pp.101-114), argues that one of the most exciting developments is the development of evolvable hardware, namely hardware that directly supports evolution-based search algorithms. Such an example is the work of Adrian Thompson (Sussex University) in Field Programmable Gate Array chips which allow real-time configuration of logic circuits that can evaluate a specified function. Also, he notes that regarding the

processing power limitations and the problem of suboptimal solutions, several distributed parallel processing techniques are investigated such as Tagging, Island Models, Parallel Implementation of GP systems, and the use of mobile agents as a low-cost method of performing the distribution of the genetic code across some parallel architecture. Furthermore, some advanced EA techniques under research are multi-objective optimisation (weighted sum approach and min-max methods), Parameter Control, Diploid Chromosomes, and Self-Adaptation.

Also, according to Ghanea-Hercock (2003, pp.115-116), there is an increasingly interest of companies in commercial applications based on evolutionary methods in order to solve engineering, scientific, and business problems in areas such as aerospace, business planning and operations research, biology and chemistry, telecommunications, entertainment and media, finance, manufacturing, medicine, and electronics and evolvable hardware. O'Neill and Brabazon (2009) provide an overview of recent patents relating to Genetic Programming which show that GP has a significant impact on real-world applications and a clear commercial potential. Patent applications that adopt GP encompass areas such as classification, signal processing, prediction, control, neural networks and spectral data, design, simulation, programming, medicine, bioinformatics, molecular biology, commerce and finance, games, and patents.

Regarding the future of Evolutionary Computation, Ghanea-Hercock (2003, pp.116-119) notes that some of the future directions in evolutionary computing are: Artificial Life and Co-evolution, Biological Inspiration of the EA systems, Developmental Biology, Adaptive Encoding and Hierarchy, Representation and Selection schemes, Mating Choice, and Parallelism. Furthermore, De Jong (2006, pp.211-230) provides a comprehensive list of current development directions where much work needs to be done to begin to include them in a more systematic fashion, such as self-adapting EAs, dynamic landscapes, parallelism exploitation, multi-objective EAs, hybrid EAs, biological inspired extensions, and evolving executable objects for the purpose of representing agent behaviours in agent-oriented domains where agents learn and/or modify their behaviours with little human intervention. About the later, Pfeifer and Scheier (2001, pp.227-229) argue that artificial evolution can be utilised for the emergence of agents behaviour, morphology, and furthermore of autonomous agent design, an approach also called evolutionary robotics. Indeed, Pfeifer and Scheier (2001, p.275) claim that evolutionary methods overcome, in

part, the biases imposed in agent designs by human designers. These biases are imposed due to the fact that implicit knowledge constrains designers' thinking.

De Jong (2006, pp.231-232) argues that a few broad themes that he thinks represent important future directions are the following: Modelling General Evolutionary Systems and More Unification of the Evolutionary Computation field. Regarding the first, De Jong states that even though biological fidelity and computational effectiveness pose conflicting EA design objectives, it is the dynamical systems perspective that evolutionary systems modellers and evolutionary computation community share and in which the later has much to contribute. In addition, EA models also have the potential for providing significant insights into complex adaptive systems. Also, De Jong argues that is hard to imagine how intelligent decisions can be made without the kind of insights the evolutionary computation community can provide.

Regarding the work toward the development of an overarching framework of the evolutionary computation field (unification of the field), De Jong (2006, p.232) states four main reasons why such a framework is critical: a) It provides a means by which "outsiders" can obtain a high-level understanding of evolutionary computation; b) it provides a means for comparing and contrasting various evolutionary computation techniques; c) it provides for a more principled way to apply evolutionary computation to new application areas; and d) a unified view of evolutionary computation serves to clarify where the gaps in our understanding lie and what the open research issues are.

Finally, O'Neill, Vanneschi, Gustafson and Banzhaf (2010) suggest ways to overcome some of the open issues of Genetic Programming in order to realise this field its full potential and to become a trusted mainstream member of the computational problem solving toolkit.

2.2 Grammatical Evolution

2.2.1 Introduction

Ryan, Collins and O'Neill (1998) introduce a new evolutionary algorithm that evolves complete computer programs, named Grammatical Evolution. This new approach uses a variable length linear genome which governs how a Backus Naur Form grammar

definition is mapped to an executable program. The result is that expressions and programs of arbitrary complexity may be evolved. Ryan, Collins and O'Neill (1998) applied their proposed system to a symbolic regression problem and the successful results lead them to the belief that the specific approach will be a huge boost for evolutionary algorithms.

Grammatical Evolution is based on the principles of three diverse fields: Evolutionary Automatic Programming, Molecular Biology, and Grammars (O'Neill and Ryan 2003, pp.1-4). Even though Grammatical Evolution is a form of Genetic Programming, it differs from the traditional Genetic Programming (Koza, 1992) in three ways (O'Neill and Ryan 2003, p.1): a) it employs linear genomes; b) it performs an ontogenetic mapping from genotype to phenotype; and c) it uses a grammar to dictate legal structures in the phenotypic space. Furthermore, O'Neill and Ryan (1999b; 1999f) argue that Grammatical Evolution is closer to natural DNA based evolutionary processes than Genetic Programming.

Regarding the grammars, O'Neill and Ryan (2003, p.2) note that their beauty is that they provide a simple mechanism that can be used for the description of any complex structure such as languages, graphs, neural networks, mathematical expressions, molecules compounds, and more. It is this feature of Grammatical Evolution which makes it a powerful tool for the evolution of any arbitrary structure as long this structure can be defined by a context free grammar.

The first application of a genotype-to-phenotype mapping in Genetic Programming is the Binary Genetic Programming invented by Banzhaf (1994). Also, the approach that a genotype is mapped to a phenotype using a BNF grammar definition has been taken already by other former evolutionary algorithms – Genetic Algorithm for Developing Software (Paterson and Livesey, 1997) – but the innovation with Grammatical Evolution is that it does not use a one-to-one mapping, and moreover it evolves individuals that contain no introns due to the mapping mechanism which does not skip codons, and the prune operator which removes genome segments not used in the genotype-to-phenotype mapping (Ryan, Collins and O'Neill, 1998). Also, Grammatical Evolution, due to its innovative mapping formula, in conjunction with a genome wrapping mechanism and the use of variable length genomes, provides a solution to the "closure" problem of Genetic Programming and grammar-based automatic programming algorithms, namely the issue of

generation and preservation of valid programs (O'Neill and Ryan, 2001). In Grammatical Evolution, any completely mapped individual (individual without non-terminal in its phenotype) is always valid.

2.2.2 Molecular Biology Influences

O'Neill (1999) notes that the aim of his work is the development of an automatic programming system drawing inspiration from biological genetic systems. That is, using techniques which simulate and reproduce evolution of genotype, many to one mapping between genotype and phenotype, degeneration, neutral mutations, and survival of the fittest.

O'Neill and Ryan (2003, pp.1-4) mention that inspiration from nature can be taken to employ not only its products but its tools as well. In this way, the field of evolutionary computation has taken stock of the power of evolution. In addition, Grammatical Evolution delves further into nature's processes at a molecular level, embraces the developmental approach and draws upon principles that allow an abstract representation of a program to be evolved. This abstraction enables GE to accomplish the following things:

- Separation of the search (genotype) and solution (phenotype) spaces.
- Evolution of programs in an arbitrary language which is described by a BNF grammar definition.
- Enabling the existence of degenerate genetic code which facilitates the occurrence of neutral mutations (many-to-one mapping, namely various genotypes can represent the same phenotype).
- Adoption of a wrapping operation during the genotype-to-phenotype mapping process that allows the reuse of the same genetic material for the production of different phenotypes.

Fundamental principles from Molecular Biology provide inspiration for Grammatical Evolution (O'Neill and Ryan 2003, pp.23-32). The most important of them is the already mentioned distinction of genotype and phenotype. Other principles that inspired GE are the genetic code and the gene expression models (Central Dogma of Molecular Biology), the Wobble Hypothesis by Crick, the Silent or Neutral mutations, the Genetic Code

Degeneracy, the Operon Model by Jacob and Monod, the Neutral Theory of Evolution, the maintenance of the Genetic Code Diversity, and the notion of actual and effective genome length (O'Neill and Ryan, 1999f).

The next section describes in detail the Grammatical Evolution algorithm and shows the correspondence of the GE system with a biological genetic system.

2.2.3 The Grammatical Evolution Algorithm

Grammatical Evolution (O'Neill and Ryan, 2001) is an evolutionary algorithm that can evolve complete programs in an arbitrary language using populations of variable-length binary strings. Namely, a chosen evolutionary algorithm (typically a variable-length genetic algorithm) creates and evolves a population of individuals and the binary string (genome) of each individual determines which production rules in a Backus Naur Form (BNF) grammar definition are used in a genotype-to-phenotype mapping process to generate a program.

In natural biology, there is no direct mapping between the genetic code and its physical expression. Instead, genes guide the creation of proteins which affect the physical traits either independently or in conjunction with other proteins (Ryan, Collins and O'Neill 1998). Grammatical Evolution treats each genotype to phenotype transition as a "protein" which cannot generate a physical trait on its own. Instead, each one protein can result in a different physical trait depending on its position in the genotype and consequently, the previous proteins that have been generated.

Figure 2.1 shows the comparison between the GE system and a biological genetic system (O'Neill and Ryan, 2001). The binary string of the genotype of an individual in GE is equivalent to the double helix of DNA of a living organism, each guiding the formation of the phenotype. In the case of GE, this occurs via the application of production rules to generate the terminals of the resulted program (phenotype) and in the biological case, directing the formation of the phenotypic protein by determining the order and type of protein subcomponents (amino acids) that are joined together.

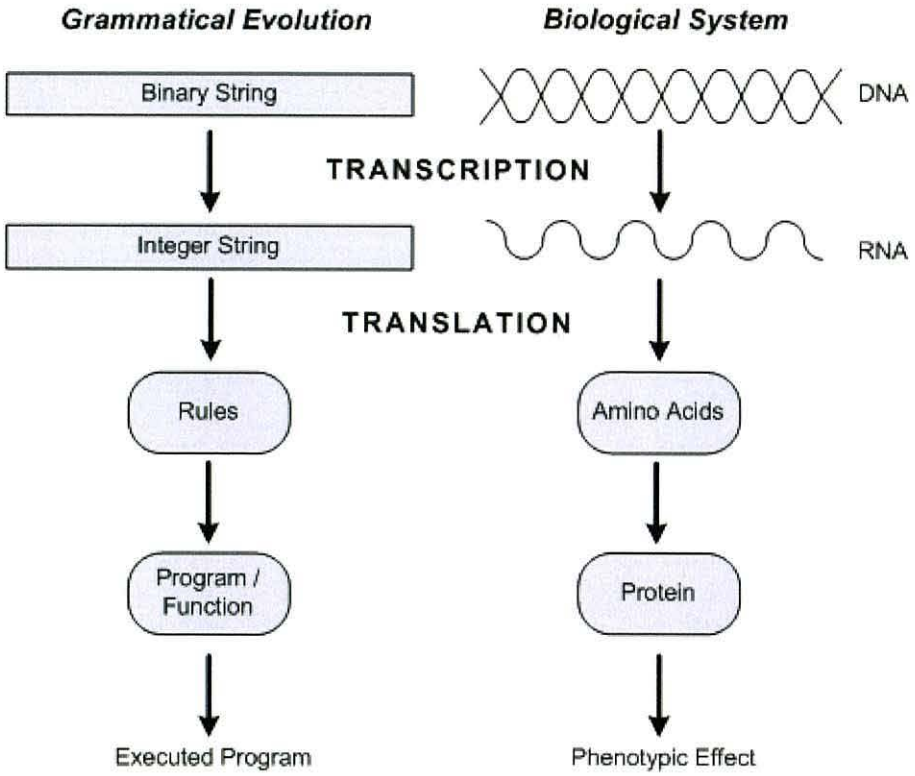


Figure 2.1: Comparison between the GE system and a biological genetic system. Cited in O'Neill and Ryan, 2001, p.351.

Before the evaluation of each individual, the following steps take place in Grammatical Evolution:

i. The genotype (a variable-length binary string) is used to map the start symbol of the BNF grammar definition into terminals. The grammar is used to specify the legal phenotypes.

ii. The GE algorithm reads "codons" of typically 8 bits (integer codons are also widely adopted) and the integer corresponding to the codon bit sequence is used to determine which form of a rule is chosen each time a non-terminal to be translated has alternative forms. If while reading "codons", the algorithm reaches the end of the genotype, it starts reading again from the beginning of the genotype (wrapping).

iii. The form of the production rule is calculated using the formula `form = codon mod forms` where `codon` is the codon integer value, and `forms` is the number of alternative forms for the current non-terminal.

An example of the mapping process employed by Grammatical Evolution is shown in Figure 2.2.

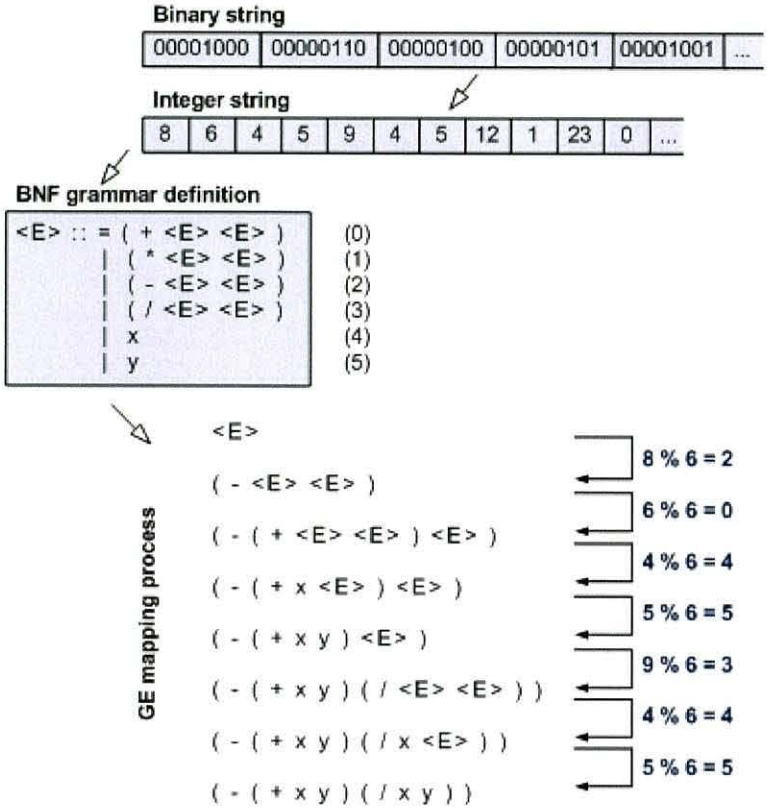


Figure 2.2: GE mapping process. Cited in Dempsey, O'Neill and Brabazon, 2006, p.2588 (with minor changes).

In this example, the first codon of the genotype of the individual is the binary string *00001000* which is the binary form of the integer *8*. The start symbol *<E>* has six alternative forms, therefore the form to be applied is this with label 2 *(8 % 6)* which results in the expression *(- <E> <E>)*. The next codon is then read in order to replace the first non-terminal symbol *<E>* of the new expression, and this goes on until the expression contains only the terminal symbols *x* and *y* and any of the arithmetic operators. Namely, until all of the non-terminal symbols have been replaced.

After the mapping process (i.e. the creation of the phenotype), the fitness score is calculated and assigned to each individual (phenotype) according to the given problem specification. These fitness scores are sent back to the evolutionary algorithm which uses them to evolve a new population of individuals.

2.2.4 Configuration of Grammatical Evolution

This section describes a general-purpose configuration of what could be called the classical Grammatical Evolution algorithm, and is based mainly on the early period from the invention of the algorithm. Since then, many variations have been applied which are mentioned in this and the subsequent sections.

In the beginning, Grammatical Evolution used a Genetic Algorithm as the search engine (O'Neill and Ryan, 2001; 2003, p.44), applying a steady-state replacement mechanism when individuals are passed from generation to generation, such that when two parents produce two children, the best of these children replaces the worst individual in the population only if it has a greater fitness than the individual to be replaced (O'Neill, Ryan, Keijzer and Cattolico, 2001). Since then many variations of the search engine have been applied (O'Sullivan and Ryan, 2002; O'Neill and Brabazon, 2006a; 2006b) with most dominant being variations of the standard Genetic Algorithm (O'Neill and Brabazon, 2005a; UCD, 2008).

The standard reproductive mechanisms of Genetic Algorithms are used in GE. A point mutation operation which acts by testing each bit locus of a genome and mutating each with a pre-specified probability, typically of value 0.01 (O'Neill and Ryan, 2001; 2003), and for recombination, a standard one-point crossover with typical probability 0.9 (O'Neill and Ryan, 2001; 2003). Namely, two crossover points are selected at random, one on each individual. Then the segments on the right hand side of each individual are swapped. O'Neill and Ryan (2000a) show that the standard one-point crossover operator is the most consistent and in general produces more successful runs than other types of crossover. O'Neill, Ryan, Keijzer and Cattolico (2001) discovered some interesting properties (e.g. ripple crossover and intrinsic polymorphism of codons) that could yield an insight into the success of the standard one-point crossover operator.

Except for the above operators, Grammatical Evolution introduces two new ones: Duplicate and Prune (Ryan, Collins and O'Neill, 1998). The gene duplication is analogous to the production of more copies of a gene or genes, in order to increase the presence of a protein or proteins in the cell of a biological organism. Duplication involves random selection of a) the number of codons to be duplicated and b) the starting position of the first codon in this set. The position of the new duplicated codons is the penultimate codon

position at the end of the chromosome. A typical probability value is 0.01 (O'Neill and Ryan, 2001; 2003).

The gene pruning discards genes not used in the genotype to phenotype process and reduces the number of introns in the genotype. According to Ryan, Collins and O'Neill (1998), pruning results in dramatically faster and better crossovers. Later research questions the usefulness of this operator because of the important role of introns (O'Neill and Ryan, 1999d; O'Neill and Ryan, 2001). For this reason, if this operator is used at all it is suggested that it be applied with a very small probability, typically 0.01 (O'Neill and Ryan, 1999d; 2001; 2003).

Regarding the codon size, the 8-bit codon is the typical size but it is not expected to be optimal across all problems (O'Neill, Ryan and Nicolau, 2001; O'Neill and Ryan, 2001; 2003, p.72). O'Neill and Ryan (2003, p.72) suggest that an investigation into more appropriate codon sizes for each problem may be beneficial. Another type of genotype representation is the integer form where each codon is represented as an integer number. Hugosson, Hemberg, Brabazon and O'Neill (2010) analyse these two representations suggesting the use of the integer variant.

For the reuse of the genetic material, a typical wrapping threshold is 10 (O'Neill, Ryan, Keijzer and Cattolico, 2001).

The typical method for the generation of the initial population is random generation of variable-length binary strings within a pre-specified range of codons, usually 1 to 10 (O'Neill and Ryan 2003, p.45). In the libGE implementation, the size of the initial random genotypes is between 15 and 25 codons (Nicolau, 2005). As an alternative initial population generation, O'Neill and Ryan (2003, pp.125-127) suggest that a sensible initialisation (analogous to GP ramped half-and-half) should be investigated. They say that this can be achieved through the grammar with recursive production rules. Initial results (O'Neill and Ryan 2003, p.127) show an improvement in the performance of GE by providing a far greater diversity than random initialisation. Harper (2010) explores three initialisation schemes (random bit, ramped half-and-half, and Sean Luke's probabilistic tree-creation version two) highlighting the importance of initialisation for the performance of an evolutionary run.

The search can be biased in Grammatical Evolution by configuring the codon length (size) and using grammar defined introns (O'Neill, Ryan and Nicolau, 2001). With grammar defined introns, the incorporation of introns into the genome is implemented through the grammar. This is achieved by allowing codons to be skipped during the mapping process, with using introns as choice for non terminals (O'Neill and Ryan 2003, p.102).

Finally, Grammatical Evolution can incorporate GP Automatically Defined Functions (ADFs) and other approaches to automatic function definition by using a grammar-based function definition (O'Neill and Ryan, 2000b). Swafford, O'Neill and Nicolau (2011) investigate modularity and extend the ADF approach in GE with the introduction of "modules" which are not modified by genetic operators and are stored in the universal grammar available to all individuals.

2.2.5 Outline of Past and Current Research

Ryan, O'Neill, and Collins (1998) introduce Grammatical Evolution and evaluate the algorithm using a trigonometric identity problem. The experimental results show that GE, like GP, is able to create its own constants. Ryan and O'Neill (1998) apply GE in three problems (symbolic regression, trigonometric identity, symbolic integration) showing that using a steady state approach regarding the applied Genetic Algorithm, improves dramatically the performance. O'Neill and Ryan (1999d) benchmark Grammatical Evolution using the Santa Fe Ant Trail problem, showing that GE outperforms GP when the later is used without the solution length fitness measure.

O'Neill and Ryan (1999e) analyse the feature of genetic code degeneracy and investigate its effects on the genotypic diversity of the population, concluding that the degeneracy of the genetic code results in improvement of the genetic diversity and in the preservation of valid individuals during runs of Grammatical Evolution. Modularity in GE is investigated by O'Neill and Ryan (2000b) with the application of grammar-based automatic function definition (ADF). The experiments demonstrate that the presence of grammar-based function definitions results in better solutions, faster than in their absence, and the finally produced code is in a more readable format. Further research on modularity in Grammatical Evolution is presented in section 2.3.3.

O'Neill and Ryan (2000c) argue that the advantages gained from the incorporation of biological inspired principles, such as genes expression, in the implementation of evolutionary automatic programming systems are the separation of search and solution spaces, the maintenance of genetic diversity, the preservation of functionality, the generalised encoding, the degenerate encoding, the compression of representation, the alternative implementations of functions, and the positional independence.

The usefulness of the crossover operator in Grammatical Evolution is investigated in O'Neill and Ryan (2000a) showing that the standard one point crossover is not as destructive as it might originally appear and that performs better than other types of crossover operators like homologous crossover and two point crossover. O'Neill, Ryan, Keijzer and Cattolico (2001) demonstrate that it outperforms also the headless chicken-like crossover. O'Neill, Ryan, Keijzer and Cattolico (2003) further analyse crossover in Grammatical Evolution, which is termed ripple crossover due to its defining characteristics. The role in Grammatical Evolution of the other genetic operator, the one-point mutation, is investigated in Rothlauf and Oetzel (2006) and Byrne, O'Neill, McDermott and Brabazon (2009).

The use of introns and their role in affecting the bias of the genetic algorithm search is investigated by O'Neill, Ryan and Nicolau (2001). The change of the codon length and the number & position of introns result in biases toward certain production rules of the grammar of the Grammatical Evolution algorithm. Also, with the performed experiments, the beneficial effect of introns for the Grammatical Evolution system is proved.

Hugosson, Hemberg, Brabazon and O'Neill (2010) analyse and compare the binary and integer representations of genotype in Grammatical Evolution to determine if one has a performance advantage over the other and they conclude that the results provide support for the use of the integer form of the genotypic representation as the binary representation do not exhibit better performance, and the integer representation provides a statistically significant advantage on one of the three benchmarks. Regarding the binary representation, McGee, O'Neill and Brabazon (2010) use a fixed-length genotype instead of the standard variable-length.

Dempsey, O'Neill and Brabazon (2005) examine the utility of meta-grammar constant creation and evolution applying Grammatical Evolution to evolve the BNF Grammar of a

Grammatical Evolution run. White, Reif, Gilbert and Moore (2005) compare Grammatical Evolution, Grammatical Swarm, and Random Search. Dempsey, O'Neill and Brabazon (2009, p.165) conduct experiments in dynamic problems and highlight GE's ability to maintain diversity within the population as a function of the potential fitness in the search space. The main issues to be solved regarding the dynamic problems are adaptability, memory, and robustness, because as they note (2009, pp.2-7), for dynamic environments the focus shifts to that of survival in a changing environment from that of attaining an optimal or perfect solution in a static environment.

Keijzer, Babovic, Ryan, O'Neill and Cattolico (2001) combine Grammatical Evolution with logic grammar in their Adaptive Logic programming System. More different grammar types have been explored with Grammatical Evolution including attribute grammars (Cleary, R., 2005), Christiansen grammars (Ortega, Cruz and Alfonseca, 2007), shape grammars (O'Neill, et al., 2009), and tree-adjunct grammars (Murphy, O'Neill, Galván-López and Brabazon, 2010).

Finally, Grammatical Evolution has been applied, since its invention, to a variety of problems and fields (see section 2.2.6) and has been used with alternative implementations of the search engine, the mapping mechanism, the genetic operators, the population initialisation, and the grammar (see section 2.2.7).

2.2.6 Applications of Grammatical Evolution

O'Neill and Ryan (1999a; 1999c) apply Grammatical Evolution in a real world problem, evolution of a caching algorithm. Genetic Programming creates algorithms that do not perform as well as those designed by humans. O'Neill and Ryan (1999a; 1999c) demonstrate the creation of a Caching Algorithm in the C programming language which is general enough so that when applied to a large cache size, a huge increase in performance is observed. The results show that Grammatical Evolution is powerful enough to extract elegant solutions.

O'Neill, Brabazon, Ryan and Collins (2001a) examine the potential of Grammatical Evolution to a real world financial problem. The goal of their experiment is the uncovering of a series of useful technical trading rules for the UK FTSE 100 stock index. The Moving Average indicator is used and the results show that Grammatical Evolution

successfully evolves trading rules with a performance superior to the benchmark buy and hold strategy. The preliminary methodology used in this experiment includes a number of simplifications because the goal is to investigate if there is any potential usefulness of using Grammatical Evolution in such problems. They argue that the results are encouraging and trigger further investigation of such applications of GE.

O'Neill, Brabazon, Ryan and Collins (2001b) further investigate the application of the Grammatical Evolution algorithm in financial problems. In their experiment, the data set is taken from the ISEQ, the official equity index of the Irish Stock Exchange. The preliminary findings of this experiment indicate that their approach has much potential. Even though the results show that the adopted model suffers from over-fitting to the training data set, however the risk involved with the adoption of the evolved trading rules is less than the benchmark buy and hold strategy.

O'Neill, Cleary and Nikolov (2004) and Cleary and O'Neill (2005) apply GE on a 0/1 Multi-Dimensional Knapsack problem (or Multi-Constrained Knapsack problem) using an attribute grammar which allows the encoding of context-sensitive and semantic information. The results demonstrate that GE in conjunction with alternative grammar representations can provide an improvement over the standard context-free grammar for problems in this domain. A detailed study is provided in Cleary (2005).

Amarteifio and O'Neill (2004) demonstrate evolutionary pattern-forming swarms using Grammatical Evolution and evolve systems that form consistent patterns through the interaction of constituent agents or particles. Namely, Grammatical Evolution is applied in a multi-agent model where information processing ants cooperate to solve an abstract clustering problem. The model considers artificial ants as walking sensors in an information-rich environment and Grammatical Evolution is combined with this swarming model as an ant's response to information is evolved. The experimental results show that the ant colony successfully evolves templates that exhibit clustering behaviour based on a spatial entropy measure.

Amarteifio and O'Neill (2005) present a novel XML implementation of GE named XMLGE with a number of features such as a rich representation approach which considers tagging information and exploiting the genotype-phenotype map; use of XSLT for genetic

operators; and the use of reflection for tree object building based on an XML expression tree (Amarteifio, 2005).

Gavrilis, Tsoulos, Georgoulas, and Glavas (2005) propose a method based on Grammatical Evolution for feature extraction and efficient classification of the Cardiotocogram (CTG) which is the most widely used Electronic Foetal Monitoring (EFM) method worldwide. The proposed method extracts new features from existing ones using nonlinear transformations and is tested on a data set of intrapartum cases with an accuracy of 92.5%.

Tsoulos, Gavrilis and Glavas (2005) present a method based on Grammatical Evolution for the construction of artificial neural networks (ANN). The method is capable of constructing both neural networks with an arbitrary number of hidden layers and recurrent neural networks. The performance of the proposed method is tested on a series of classification and regression problems and is compared against a traditional Multilevel Perceptron trained with the Tolmin optimisation method giving better results.

Brabazon and O'Neill (2006) apply Grammatical Evolution to computational finance which presents significant real-world challenges to machine learning arising from complexity, noise, and constant change. In particular, they apply GE to develop rules for trading systems and in traditional classification problems such as bond rating and predicting corporate failure.

O'Neill and Brabazon (2008) apply Grammatical Evolution to evolve a logo design for the UCD Natural Computing Research & Applications group. A Lindenmayer system (L-System) is evolved which is expressed in the postscript language and a human is acting as the fitness function of the generated L-Systems.

Dempsey, O'Neill and Brabazon (2009, pp.141-161) demonstrate the application of Grammatical Evolution to financial trading. They simulate live trading over historical financial time series and produce rules that outperform a benchmark buy-and-hold strategy in 23% of runs conducted on the Nikkei 225 data. Competitive results are also produced for the S&P 500 Index.

Some of the recent applications of Grammatical Evolution involve interactive generative music synthesis (Shao, McDermott, O'Neill and Brabazon, 2010), quadrupedal animal animation (Murphy, O'Neill and Carr, 2009; Murphy, Carr and O'Neill, 2010; Murphy, 2011), the evolution of robocode tanks (Harper, 2011), the evolution of a Ms. PacMan controller (Galván-López, Swafford, O'Neill and Brabazon, 2010), the generation of 3D images (Nicolau and Costelloe, 2011), evolutionary architectural design (Byrne, et al., 2011), the evolution of controllers for the Mario AI Benchmark (Perez, Nicolau, O'Neill and Brabazon, 2011), the evolution of dynamic trade execution strategies (Cui, Brabazon and O'Neill, 2010), and the femtocell coverage of telecommunication networks (Hemberg, Ho, O'Neill and Claussen, 2011).

2.2.7 Grammatical Evolution Variations

Nicolau and Ryan (2003) describe GAuGE (Genetic Algorithms using Grammatical Evolution). GAuGE is based on the principles of Grammatical Evolution and encodes both the value and the position on each gene, in order to become a position-independent genetic algorithm. Specifying the position and the value of each gene may result in grouping together important data in the genotype string. In this way the breaking and disruption during the evolution process is prevented. The experiments show that GAuGE is moving the more salient genes to the start of the genotype string, creating in this way robust individuals that are built in a progressive fashion from the left to the right side of the genotype.

Nicolau and Ryan (2002) introduce LINKGAUGE which is an extension of the GAuGE algorithm. The LINKGAUGE tackles the class of deceptive linkage problems by using a simple and extremely effective algorithm in terms of required computer memory. The results of the experiments show that LINKGAUGE has a very interesting scale-up property that it is much less demanding than the original messy Genetic Algorithm in hardware resources (memory and over-head processing power).

O'Neill and Ryan (2003; pp. 99-128) present and discuss some variations and extensions of Grammatical Evolution, like the implementation of an alternative translation function (the Bucket Rule), the incorporation of grammar defined introns, a sensible initialisation using recursive production rules in the grammar, and the Chorus algorithm which is a position independent algorithm based on the principles of the cells metabolism.

Chorus (Ryan, O'Neill and Azad, 2001; Ryan, Azad, Sheahan and O'Neill, 2002; Azad, 2003) uses a chromosome of eight bit numbers (termed genes instead of codons) to dictate which rules from the BNF grammar to apply. However, unlike the intrinsically polymorphic codons of GE, each gene in Chorus corresponds to a particular production rule. The modulo is taken with the total number of production rules in the grammar, consequently a gene always represents a particular rule regardless of its position in the chromosome (the meaning of a gene is determined by those that precede it).

O'Neill, Brabazon and Adley (2004) examine the application of Grammatical Swarm to classification problems. Grammatical Swarm adopts a Particle Swarm learning algorithm coupled to a Grammatical Evolution genotype-to-phenotype mapping to generate computer programs. Each particle (or real-valued vector) represents choices of program construction rules specified as production rules of a Backus-Naur Form grammar. This variation is tested in a mushroom classification problem and a bioinformatics problem (detection of eukaryotic DNA promoter sequences). For the first problem the generated solutions take the form of conditional statements in a C-like language subset and for the second problem the solutions take the form of simple regular expressions. The results of these experiments show that it is possible to create computer programs using the Grammatical Swarm technique and that the performance of this technique is similar to the performance of the Grammatical Evolution approach. The key finding is that Grammatical Swarm is able to generate solutions to problems of interest even though the Swarm algorithm is relatively simple, the sizes of the involved populations are small, the length of the vector representation is fixed, no crossover operator is used, there is no concept of selection or replacement, and the parameters of the algorithm had not been optimised.

O'Neill and Ryan (2004) introduce Grammatical Evolution by Grammatical Evolution ($(GE)^2$), a novel algorithm which evolves not only the genotype of the individual but the BNF grammar definition as well. $(GE)^2$ uses a diploid chromosomal structure. The one chromosome describes the solution and the other chromosome is the individual's own grammar (solution grammar) which is used to map the solution chromosome. A meta-Grammar (universal grammar) is used to map the grammar chromosome of each individual.

O'Neill and Brabazon (2005b) also present meta-Grammar GA (mGGA). Meta-grammars are used in the development of the mGGA that encourages the evolution of building blocks which can be reused to construct solutions more efficiently.

Harper and Blair (2005) discuss a new type of crossover for Grammatical Evolution, the LHS Replacement Crossover. It uses information automatically extracted from the grammar to minimise any destructive impact from the crossover. The information is extracted the same time the genotype is initially decoded and allows the swapping between entities of complete expansions of non-terminals in the grammar without disrupting useful blocks of code on either side of the two point crossover. The experimental results demonstrate that the LHS Replacement Crossover provides substantial benefits over all other examined crossover types at least in the examined domains (it is the most successful in exploring the search space). Indeed, the results confirm that in the explored domains, the proposed crossover operation is more productive than hill-climbing, it enables populations to continue to evolve over considerable numbers of generations without intron bloat, and it allows populations to reach higher fitness levels quicker.

Continuing their work on Grammatical Evolution crossover operator, Harper and Blair (2006a) propose a self-selecting crossover operator (variable crossover) and compare its efficacy in a typical numeric regression problem and a typical data classification problem. They propose a mechanism which allows the evolutionary algorithm to self-select the type of crossover utilised and which is shown to improve the rate of generating 100% successful solutions. They compare their proposals against a one-point crossover, a LHS (Left Hand Side) crossover, and a random crossover (which selects the crossover type randomly). Their motivation is to attempt to find a crossover operator that represents a "good choice" in the absence of any empirical evidence as to the best crossover operator to use. The experiments show that the variable crossover operator performs substantially better than either the one-point or LHS on their own, and at least as well as, and in some cases significantly better than, the alternative strategy of randomly selecting a crossover operator to apply. They conclude that given that the variable crossover operator did not perform significantly better than the random crossover operator in one of the domains examined, further work needs to be done in relation to verifying that the variable

crossover operator generalises and in particular with respect to problem domains which favour either the one-point crossover or LHS crossover.

Harper and Blair (2006b) introduce a meta-grammar into Grammatical Evolution allowing the grammar to dynamically define functions, self-adaptively at the individual level without the need for special purpose operators or constraints. The user does not need to determine the architecture of the dynamically defined functions (which is the case in automatically defined functions). As the search proceeds through genotype/phenotype space, the number and use of the functions can vary. Harper and Blair argue that the ability of the grammar to dynamically define such functions allows regularities in the problem space to be exploited even where such regularities were not apparent when the problem was set up.

Ošmera, Popelka and Pivoňka (2006) describe a variation of the Grammatical Evolution algorithm, named Parallel Grammatical Evolution (PGE), which maps genotype to phenotype with backward processing (the processing of the production rules is done backwards, from the end of the rule to the beginning) and uses a novel variation of a Hierarchical Parallel GA as the search engine. Namely, the population is divided into several sub-populations that are arranged in the hierarchical structure. Every sub-population has two separate parts: a "male" group and a "female" group. Every group uses quite a different type of selection (the first group uses a classical type of GA selection). Indeed in the second group only different individuals can be added. They argue that PGE has proved successful for creating trigonometric identities, and that PGE with hierarchical structure can increase the efficiency and robustness of systems, and thus they can track better optimal parameters in a changing environment. From the experimental session they conclude that modified standard GE with only two sub-populations can create PGE much better than classical versions of GE.

Nicolau and Dempsey (2006) introduce two grammar-based extensions for Grammatical Evolution. The first of them is `<GECodonValue [-x1] [+x2]>` which allows the direct phenotypic use of genotypic values. The second extension is `<GEXOMarker>` which uses information from the mapping process to setup a list of crossover locations in the genotype string. These extensions attempt to "blur" somehow the genotype/phenotype

separation allowing some information to be shared between the search and the solution spaces.

Ortega, Cruz and Alfonseca (2007) describe Christiansen Grammar Evolution, an extension of Grammatical Evolution that improves the expressiveness power of the later. This new automatic programming algorithm uses for the description of legal phenotypes a Christiansen grammar instead of a context-free grammar. While Grammatical Evolution uses a context-free grammar, expressed in the BNF notation, to avoid syntactic mistakes of the generated phenotype, the Christiansen Grammar Evolution algorithm uses an adaptive extension of attribute grammars in order to create not only syntactically but also semantically correct phenotypes according to some predefined semantic conditions.

TAGE (Murphy, O'Neill, Galván-López and Brabazon, 2010; Murphy, 2011) is a Grammatical Evolution variation which uses a tree-adjunct grammar (TAG) instead of a context free (CFG). TAGs are more powerful than CFGs (Joshi and Schabes, 1997), which are currently used in standard GE, since the set of languages produced by TAGs is a super-set of those produced by CFGs. Unlike CFGs, TAGs can also generate some context-sensitive languages. In addition to this, it has been shown that for every CFG there is a TAG that is both weakly and strongly equivalent to it (Joshi, 1985). Because of the use of TAGs, derivation in TAGE is different from derivation in GE in that it is a two step process, first a derivation tree is formed, and from that the derived tree is produced.

Finally, Dempsey, O'Neill and Brabazon (2009, pp.9-24), mention two more GE variations: Position Independent GE (πGE) and Grammatical Differential Evolution (GDE). The first variation (πGE), removes the positional dependency of standard GE where the mapping process moves from left to right in consuming the non-terminals. Codons in πGE have two values: *nont* which contains the encoding to select which non-terminal is to be consumed by the mapper, and *rule* which contains the encoding to select the production rule for the selected non-terminal. The other variation (GDE) adopts a Differential Evolution learning algorithm coupled to Grammatical Evolution genotype-to-phenotype mapping to generate programs in an arbitrary language.

2.2.8 Issues and Future Work

Grammatical Evolution has been shown to be an effective and efficient evolutionary algorithm in a series of both static and dynamic problems (Dempsey, O'Neill and Brabazon, 2009). However, there are known issues that still need to be tackled such as:

1. Destructive crossovers (O'Neill, Ryan, Keijzer and Cattolico, 2003).
2. Genotype bloating (Harper and Blair, 2006b).
3. Dependency Problems (Ryan, Collins and O'Neill, 1998).
4. Low locality of genotype-to-phenotype mapping (Rothlauf and Oetzel, 2006).

Destructive crossover events and bloating are generally regarded to be GP and GE issues (O'Neill, Ryan, Keijzer and Cattolico, 2001; 2003; Hemberg, 2010). But suggestions have been made (O'Neill, Ryan, Keijzer and Cattolico, 2003) that these two issues are interrelated in the way that destructive crossover events could be responsible for bloat, arising as a mechanism to prevent destructive crossover events occurring by acting as buffering regions in which crossover can occur without harming functionality.

Indeed, Grammatical Evolution is, like Genetic Programming, subject to problems of dependencies (Ryan, Collins and O'Neill, 1998). For example, the further a gene is from the root of the genome, the more likely it will be affected by the previous genes. Ryan, Collins and O'Neill (1998) suggest the biasing of individuals to a shorter length and the progressive generation of longer genomes.

Rothlauf and Oetzel (2006), show that the representation used in Grammatical Evolution has problems with locality because in some cases (less than ten percent of the time) neighbouring genotypes do not correspond to neighbouring phenotypes. Experiments with a simple local search strategy reveal that the GE representation leads to lower performance for mutation based search approaches in comparison to standard GP representations. The results suggest that locality issues should be considered for further development of the representation used in GE. Byrne, O'Neill, McDermott and Brabazon (2009) analyse further the behaviour of mutation in GE investigating three different types of mutation operators (nodal, structural, and integer mutation) and found that nodal mutation would be beneficial in fine tuning a solution and that structural mutation acts as a technique for a more global exploration of the search space.

Besides the above known issues of Grammatical Evolution, Robilliard, et al. (2006) mention and investigate methodological problems associated with the Santa Fe Trail (SFT) problem. They discuss the difficulty to ensure fair comparison especially with new genotype representations as found in works on grammar-based automatic programming, such as Grammatical Evolution (GE), and Bayesian Automatic Programming (BAP). They show that the setups of the published SFT experiments with GE and BAP, include small changes in the benchmark definition, having great consequences in solving the problem, up to the point that comparison with GP is called into question. Indeed, they mention that the search space defined by the grammar used in GE is not semantically equivalent with the search space of the original Santa Fe Trail problem definition (namely, the grammar used by GE in the benchmark states a declarative language bias). They argue that when tackling Santa Fe Trail, from a problem-oriented point of view, it is enough to preserve the semantics of programs (same language bias), whatever the biases introduced by the representation (search bias).

O'Neill, Vanneschi, Gustafson and Banzhaf (2010) outline some of the open issues in the wider field of Genetic Programming – which must be addressed for GP to realise its full potential and to become a trusted mainstream member of the computational problem solving toolkit – and in some cases they suggest ways to overcome them.

Regarding possible lines of research that could be profitable to GE, O'Neill and Ryan (2003, pp.129-132) provide the following areas as the suggestions.

a) The Grammar (e.g. investigation of context sensitive, attribute, and logic grammars, adoption of dynamic grammars that are open to the process of evolution).
b) The Transcription Model (e.g. different reading frames after a wrap event).
c) The Translation Model (e.g. bucket rule, etc.).
d) The Evolutionary Algorithm Engine (e.g. inclusion of concepts such as diploidy and polygenic inheritance etc.).
e) The Initialisation Strategy (e.g. sensitive initialisations).
f) Neutral Evolution (investigation of the benefit of neutral evolution in GE).
g) Application Areas (e.g. investigation for what classes of problems the GE is useful).

Furthermore, Dempsey, O'Neill and Brabazon (2009, pp.163-169), mention three opportunities for future research about Grammatical Evolution in dynamic environments: a) exploration of the existence of memory within meta-Grammars and $(GE)^2$; b) examination of the potential of the solution-grammar chromosomes to evolve useful building blocks; and c) examination of the application of πGE to dynamic problems.

Dempsey, O'Neill and Brabazon (2009, p.169) note the need for further application of GE to dynamic real-world problems because this may focus research on issues in those domains, rather than on limitations of catch-all benchmark problems.

2.3 Modularity

2.3.1 Introduction

O'Neill, Vanneschi, Gustafson and Banzhaf (2010) note that individuals in classical GP are usually constructed from a primitive set of functions and terminals and that an extension to this approach is the ability to define modules which is a very important means of improving GP expressiveness, code reusability and performance. These modules are tree-based representations defined in terms of the primitives and past research show that the use of modularity in GP has helped overcome some problems that classical GP could not in a fixed number of runs or has helped solve them more efficiently (O'Neill, Vanneschi, Gustafson and Banzhaf, 2010). Furthermore, there has been a large number of studies focusing on modularity in Evolutionary Algorithms of which a brief overview can be found in Hemberg (2010, pp.67-76).

2.3.2 Modularity in Genetic Programming

The most well known of the modularity methods in Genetic Programming is Koza's *Automatically Defined Functions* or ADFs (Koza, 1992). This is an extension to standard GP which copes with the automatic decomposition of a solution function. Each individual has a fixed number of components (functions) to be automatically evolved, having a fixed number of parameters, and which result in producing branches. These components (functions) represent fragments of code playing a role of reusable subroutines which are subject to genetic operations but are not shared between individual programs. Koza (1994) proves in many examples that the main advantages of this approach are generality,

flexibility, and performance. GP with ADFs automatically discovers how to decompose a problem into subproblems, how to solve the subproblems (ADFs), and how to combine the solutions to subproblems in higher level ADFs and program body. Rosca and Ballard (1994) note that theoretically, GP extended with ADFs is not more powerful than standard GP but it is more efficient due to two main differences which are that GP with ADFs develops much more complex programs and that it is able to make larger jumps in the search space. Furthermore they argue that ADFs may have no clear meaning from the point of view of the problem solved and that they are not explicitly associated to problem subgoals even in the case when it is known what a problem subgoal is.

Another approach to modularity is the *Module Acquisition* of Angeline and Pollack (1993) which is based on the creation and administration of a library of modules which extend the problem representation. This approach uses two new genetic operators, *compress* and *expand* (Angeline and Pollack, 1994) that control the modification of the individuals. A module is a function with a unique name created by selecting and chopping off branches of a subtree selected randomly from an individual. Namely, the *compress* operator randomly chooses a node in a program tree and extracts a module from it. Compression freezes possibly useful genetic material, by protecting it from the destructive effect of other genetic operators. The *expand* operator implements the inverse function of compress. Unlike ADFs, this approach cannot reuse arguments in the modules.

Rosca and Ballard (1994) propose a bottom-up approach to modularity introducing *Adaptive Representations* and show that in GP, useful genetic material can be discovered and used globally to extend the problem representation in a hierarchical way in order to improve performance. They use the term *building blocks* to define entire subtrees of a given maximum height from population individuals and the process of discovery is based on the analysis and tracking of building blocks over generations of evolution. These blocks can be evaluated with several methods and the fittest dynamically extend the function set. When at least such a building block is discovered in a generation, the population undergoes substantial changes (*epoch*). Namely, the most fit individuals are retained and the other are replaced by new individuals randomly generated using the extended function set.

Whigham (1995a; 1995b; 1996) uses a context free grammar to define the structure of the initial language and modifies the grammar as the evolution proceeds by discovering and

adding new productions as an example of learnt bias. Individuals are represented as derivation trees from the grammar and a form of *encapsulation* builds new terms into the initial language. Productions are discovered with propagation of a terminal up the program tree to the next level of non-terminals and they are assigned with a merit value which represents the probability of selection from some non-terminal. In the case where all siblings of a propagated terminal are also terminal symbols, the terminals are encapsulated as one functional symbol. Individuals based on the modified grammar are introduced into the population with an *epoch replacement* (Rosca and Ballard, 1994) approach.

Two more recent approaches to modularity are *Run Transferable Libraries* (Keijzer, Ryan and Cattolico, 2004; Ryan, Keijzer and Cattolico, 2004) and *DEVTAG* (McKay, Hoang, Essam and Nguyen, 2006). Run Transferable Libraries is a mechanism to pass knowledge acquired in one GP run to another using libraries of useful functions. A system using RTL accumulates information from run to run, learning more about the problem each time it gets applied and updating the library over time based on past experience (after each individual run, the library is updated using statistics about the use of the library elements). DEVTAG applies developmental evaluation using Tree Adjoining Grammar Guided Genetic Programming (TAG3P) for evaluating an individual on multiple problems at different stages of development in order for the modular structure to provide an adaptive advantage to that particular individual and hence being selected by the natural selection process of evolution. DEVTAG uses fitness ordering of individuals which is generated with a special multi-stage comparison. Corresponding to the insight that later-stage fitness is only important if the individual survives earlier stages, DEVTAG compares individuals on simpler problems and only if they are roughly equivalent evaluates them on more complex problems.

Another approach to modularity is this of Majeed and Ryan (2007) who investigate the problem of identification and subsequent reuse of useful modules in GP by applying context sensitive evaluation of subtrees (contribution of the subtree in the container tree) and context-aware genetic operators (crossover and mutation). Context-aware crossover places a randomly selected subtree from one parent, in its best possible context in the other parent. Context-aware mutation operates by replacing existing subtrees with modules from a previously constructed repository of probably useful subtrees.

2.3.3 Modularity in Grammatical Evolution

The first application of modularity in Grammatical Evolution is presented in O'Neill and Ryan (2000b) where functions are defined in the grammar by predetermining their number and their parameters similarly to ADFs. Harper and Blair (2006b) extend this work and introduce a dynamic grammar approach into Grammatical Evolution allowing the grammar to dynamically define functions (DDFs), self adaptively at the individual level without the need for special purpose operators or constraints. This allows the specification of multiple functions and a variable number of parameters for each function. These functions (DDFs) are dynamically appended to the core grammar in order to be invoked by the main function.

A meta-grammar approach to modularity has also been studied. Grammatical Evolution by Grammatical Evolution or $(GE)^2$ evolves the grammar that Grammatical Evolution uses to specify the construction of a syntactically correct solution (O'Neill and Ryan, 2004). In mGGA (O'Neill and Brabazon, 2005b), the meta-grammar approach is shown to be an effective method as an alternative binary string Genetic Algorithm through the provision of a mechanism to achieve modularity. Hemberg, O'Neill and Brabazon (2009) apply $(GE)^2$ and ADFs for the dynamic definition of modules with fixed module signatures.

Swafford and O'Neill (2010) define *module* as "any sub-derivation tree or group of sub-derivation trees in an individual" and investigate Grammatical Evolution modularity in evolutionary design problems (evolution of floor plan designs) by analysing how different grammars of varying levels of modularity may be capable of producing the same phenotypes but displaying differences in performance on the same problems. They show that increase in modularity, brought about by simple modifications in the grammar, results in the increase of the quality of solutions as well.

Swafford, O'Neill and Nicolau (2011) examine a grammar-based approach to modularity by modifying the GE grammar over the course of an evolutionary run with the addition of modules as productions and using context-sensitive crossover and mutation which operate on derivation trees created by the GE genotype-to-phenotype mapping process. Swafford, et al. (2011) extend this work with modifying each individual's genotype (genotype repair) to prevent the disturbance in fitness that can come with modifying GE's grammar definition during an evolutionary run with previously discovered modules.

Another approach to modularity in Grammatical Evolution can be found in McDermott, et al. (2010) with the application of higher-order functions in aesthetic EC encodings. Also, the application of tree-adjunct grammars to Grammatical Evolution (Murphy, O'Neill, Galván-López and Brabazon, 2010) show that since the building blocks in TAGE (elementary trees) are larger than those employed by GE individual symbols, this allows access to new kinds of tree transformations that are not readily available with standard Grammatical Evolution.

2.4 Grammatical Evolution Implementations

2.4.1 libGE

The libGE library (Nicolau, 2005) is the first publicly available and open source implementation of the Grammatical Evolution system. It is written in the C++ language and has been developed under GNU/Linux. The latest stable version is the 0.26, released at 14 September 2006. The characteristics of libGE are presented in Nicolau (2006a) and O'Neill and Ryan (2001).

libGE implements the Grammatical Evolution mapping process. The library can be used by any compatible implementation of an evolutionary algorithm in order to map the genotype (the result of the search algorithm) to the phenotype (the program to be evaluated). As Nicolau (2006b) says in the documentation of libGE "On its default implementation, it maps a string provided by a variable-length genetic algorithm onto a syntactically-correct program, whose language is specified by a BNF (Backus-Naur Form) context-free grammar."

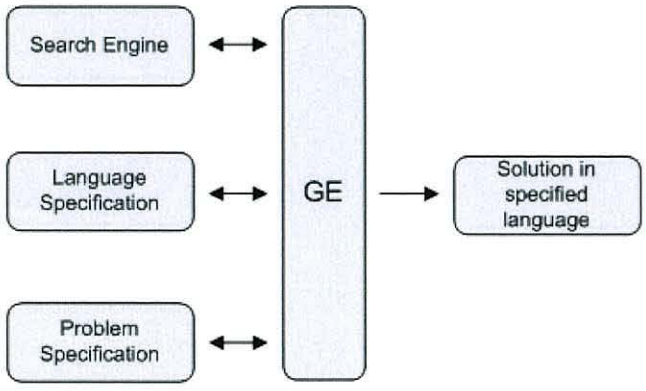


Figure 2.3: Life-Cycle of a Grammatical Evolution run. Cited in Nicolau, 2006b, p.3.

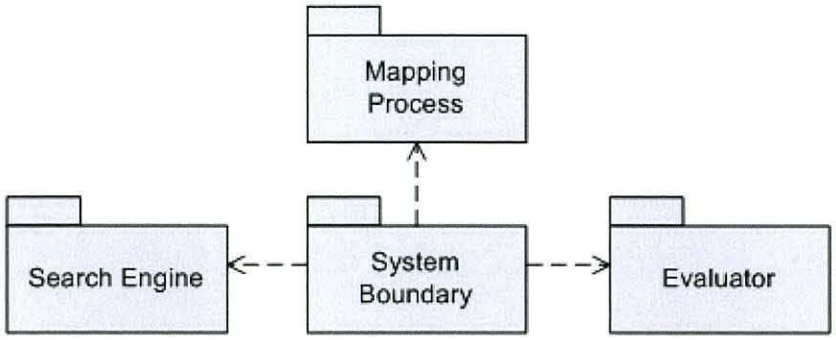


Figure 2.4: The concept of a Mapper in libGE. Cited in Nicolau, 2006b, p.6.

Figure 2.3 and Figure 2.4 illustrate the life-cycle of a Grammatical Evolution run and the concept of a *Mapper* in libGE.

Namely, the chosen *Search Engine* provides to libGE, binary strings of individuals. The strings are then mapped into programs with the use of the language specification (typically a BNF grammar). Then, the resulting programs are evaluated by the *Evaluator* which implements a problem specification (e.g. an interpreter or compiler). Finally, the result of the evaluation (the fitness score) is returned to the *Search Engine* in order to create the offspring and to feed again libGE with new individuals (binary strings).

The pseudo-code in Listing 2.1 illustrates the use of the libGE *Mapper*. With asterisk (*) are notated the steps where libGE is involved.

```
Create global mapper; *
Initialise mapper; *
Do evolutionary run
    Generate population of individuals;
    Evaluate each individual
        Transform individual to Genotype structure;
        Call Mapper.setGenotype(individual); *
        Call Mapper.getPhenotype(); *
        Pass Phenotype structure to evaluator;
        Collect fitness score from evaluator;
        Pass fitness back to search engine;
    Done
Done
```

Listing 2.1: Sample of using the libGE Mapper. Cited in Nicolau, 2006b, p.7.

The *System Boundary* is where the various data structures of libGE are interfaced. Also, libGE provides various methods which make easier the transformation of the data

structures of any search engine into libGE genotype structures. In addition, native support for data structures of some search engines like MIT GALib is implemented and included in the libGE library.

libGE was initially designed to work with Context-Free Grammars (CFG) which means that a specific codon value will always result in a specific production for a non-terminal symbol, regardless of the context in which it appears (Nicolau, 2005). Consequently, Context-Sensitive Grammars and Attribute Grammars are not supported by libGE (Nicolau, 2005).

Finally, beyond the standard Grammatical Evolution mapping process, the libGE library introduces a set of commands that can be incorporated into grammars in order to be improved the control over the mapping process. Such an example is the use of `<GECodonValue>` symbol which allows the user to insert codon values directly from the genotype into the resulting phenotype (Nicolau, 2005).

2.4.2 GEVA

GEVA (Grammatical Evolution in Java) is a publicly available and open source implementation of Grammatical Evolution in the Java programming language, developed at UCD's Natural Computing Research & Applications group (UCD, 2008). It was first released at September 2008 and the current version is the 2.0, released at June 2011.

GEVA, in contrast to libGE, is a complete implementation of the Grammatical Evolution system. Namely, it implements the Grammatical Evolution genotype-to-phenotype mapping process, a variable-length integer encoding evolutionary algorithm search engine, and a graphical user interface (GUI). Furthermore, it provides implementations of demonstration problems such as artificial ant (Santa Fe Trail, Los Altos, San Mateo), symbolic regression, L-systems, even-5-parity, royal tree, and max problem.

The software comes in two main components: GUI and GEVA. The GUI component provides a simple user interface to assist a new user to use it for simple tasks, like the configuration and execution of the provided demonstration problems (see Figure 2.5). The GEVA component is the core library and provides a command line interface to allow for scripting.

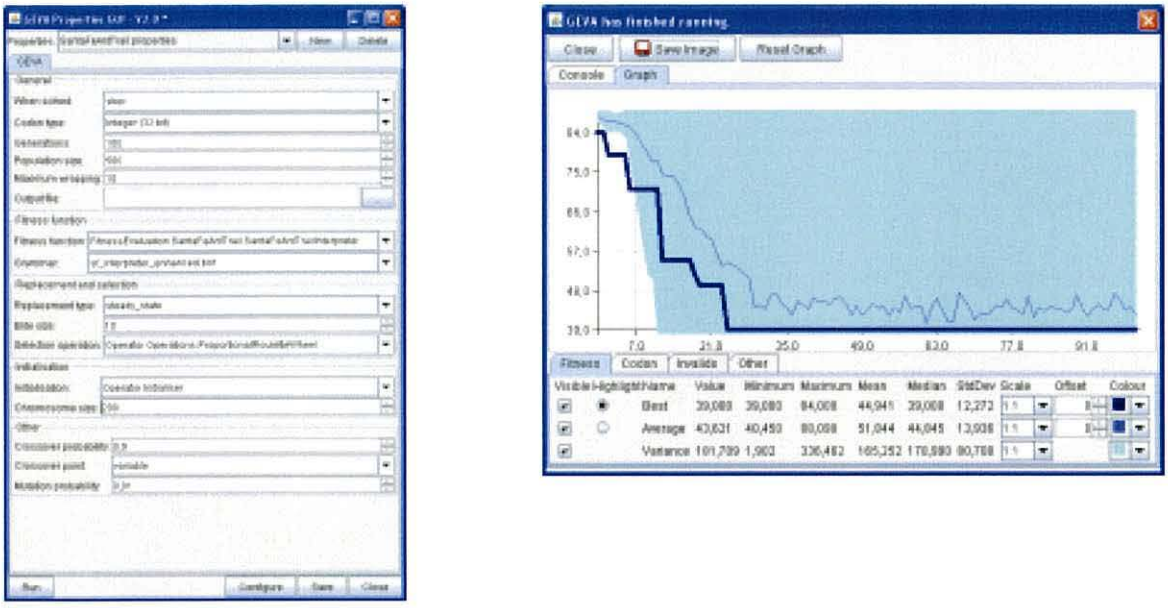


Figure 2.5: GUI component of GEVA v2.0.

It is worth noting that GEVA was not the first implementation of GE for Java – this was in fact jGE, which will be described in Chapter 3. Also, in comparative experiments to validate jGE against GEVA, the author found that the first version of GEVA (v1.0) had two bugs in the implementation of the Santa Fe Trail problem. Namely, the ant was initially facing south instead of east and the ant turned left when commanded to turn right and vice versa. These bugs were fixed in v1.2, released at July 2010 and this was credited in the change log of the program.

2.4.3 Other Implementations

Except libGE and GEVA, more implementations of Grammatical Evolution have been developed in a variety of programming languages and for various purposes. A brief description of some of them is following.

PonyGE (code.google.com/p/ponyge)

It is a small (just one source file) but functional implementation of GE in the Python language. According to its creator, it is intended as an advertisement and a starting-point for those new to GE, a reference for implementers and researchers, a rapid-prototyping medium for the creator's own experiments, and a Python workout.

PyNeurGen (pyneurgen.sourceforge.net)

Python Neural Genetic Hybrids (PyNeurGen) is a collection of libraries for use in Python programs to build hybrids of neural networks and genetic algorithms and/or genetic programming. The particular flavour of genetic algorithms/programming used is Grammatical Evolution.

DRP (drp.rubyforge.org)

Directed Programming (DRP) is a new generative programming technique in Ruby which is a generalisation of Grammatical Evolution allowing one not only to do GE, but also to do Genetic Programming in pure Ruby without explicitly generating a program string as output.

GERET (github.com/bver/geret)

Grammatical Evolution Ruby Exploratory Toolkit (GERET) is a Ruby toolkit and library designed to explore the potential of Grammatical Evolution. It provides various GE mapping implementations (bucket rule, positional independence, several node expansion strategies, wrapping), various genetic operators (ripple & LHS crossover, nodal/structural mutations), various search algorithms (SPEA2, NSGA2, ALPS), and supports attribute grammars (mapping with semantics of context-free grammars).

GEM (ncra.ucd.ie/GEM/GEM-v0.2.tgz)

Grammatical Evolution in MATLAB (GEM) is an implementation of Grammatical Evolution for the MATLAB environment. It has been developed at UCD's Natural Computing Research & Applications group by Erik Hemberg (UCD, 2008). GEM's current version is v0.2 which is distributed under the terms of the GNU General Public License.

The above implementations (except GEM) are mentioned in the Grammatical Evolution official web site (www.grammatical-evolution.org).

2.5 Evaluating Evolutionary Algorithms

2.5.1 Introduction

The performance of evolutionary algorithms and specifically of Genetic Programming and Grammatical Evolution is measured in a series of standard benchmarking problem domains such as symbolic regression (Koza, 1992; O'Neill and Ryan, 2003), symbolic integration (Koza, 1992; O'Neill and Ryan, 2003), trigonometric identity (Koza, 1992; Ryan and O'Neill, 1998) caching algorithms (Paterson and Livesey, 1997; O'Neill and Ryan, 2003), artificial ant (Koza, 1992; Koza, 1994; O'Neill and Ryan 2001; O'Neill and Ryan, 2003), and maze searching (Sondahl, 2005; Georgiou and Teahan, 2011).

These problem domains, amongst others, serve as a medium for the evaluation and comparison of various evolutionary algorithms and the investigation of different aspects and characteristics of them.

2.5.2 Symbolic Regression Problems

Symbolic regression problems involve finding a mathematical expression in symbolic form that represents a given set of input and output pairs of data. The aim of the problem is to determine the function that maps the input values onto the output values (Koza 1992, p. 11; O'Neill and Ryan 2003, p.49).

Koza (1992, p.11) notes that the mathematical expression to be found in symbolic function identification can be viewed as a computer program that takes the values of the independent variable x as input and produces the values of the dependent variable $f(x)$ as output.

A standard target function sought in the regression problem is the quadratic polynomial $x^4 + x^3 + x^2 + x$ when Genetic Programming and Grammatical Evolution are being evaluated, (Koza 1992, p.164; O'Neill and Ryan 2003, p.49).

2.5.3 Symbolic Integration Problems

Symbolic integration involves finding the mathematical expression that is the integral, in symbolic form, of a given curve (Koza 1992, p.13; O'Neill and Ryan 2003, p. 52).

Similarly to symbolic regression, a set of input and output pairs is given and the function must be found that maps one value of the pair onto the other value of the pair.

The symbolic integration problem can be reduced to symbolic regression by integrating the function examined and performing symbolic regression on the target integral curve (O'Neill and Ryan 2003, p.52). Koza (1992, p.13) notes that the mathematical expression being sought can be viewed as a computer program that takes each of the random values of the independent variable as input and produces the value of the numerical integral of the unknown curve as its output.

A standard curve being evaluated is $Cos(x) + 2x + 1$ for the integration problem in Genetic Programming (Koza 1992, p.259) and Grammatical Evolution (O'Neill and Ryan 2003, p.52) The integral in symbolic form of this curve is the mathematical expression $Sin(x) + x + x^2$ (Koza 1992, p.259; O'Neill and Ryan 2003, p.52).

2.5.4 Artificial Ant Problems

The Artificial Ant problem was devised by Jefferson, et al. (1992) with the Genesys/Tracker System. The Genesys/Tracker system was built by the UCLA Artificial Life group in a study to test the feasibility of using evolution to design programs with complex behaviour. The trail used in the original system was the John Muir Trail (Jefferson, et al., 1992).

The objective of the Artificial Ant problem (Koza 1992, p.54) is to find a computer program to control an artificial ant, so that it can find, within a reasonable amount of time, all pieces of food lying along an irregular trail located on a plane grid. The artificial ant can perform three primitive actions: *move*, *turn right*, and *turn left*. The first action moves the ant forward one square in the direction it is currently facing. When the ant moves into a square, it eats the food, if there is any. The other two actions turn the ant right or left respectively by 90 degrees, without moving the ant. Each of these actions takes one time unit. In addition, the artificial can use a sensing operation *food ahead*, which comes without cost in time units. This function looks into the square the ant is facing and returns true or false depending upon whether the square ahead contains food or is empty, respectively.

The detailed description of two instances of the Artificial Ant problem, the Santa Fe Trail and the Los Altos Hills, is following.

Santa Fe Trail

The Santa Fe Trail (Figure 2.6) is a standard problem which is used for benchmarking in Genetic Programming (Koza, 1992; Koza, 1994) and Grammatical Evolution (O'Neill and Ryan, 2001; O'Neill and Ryan, 2003).

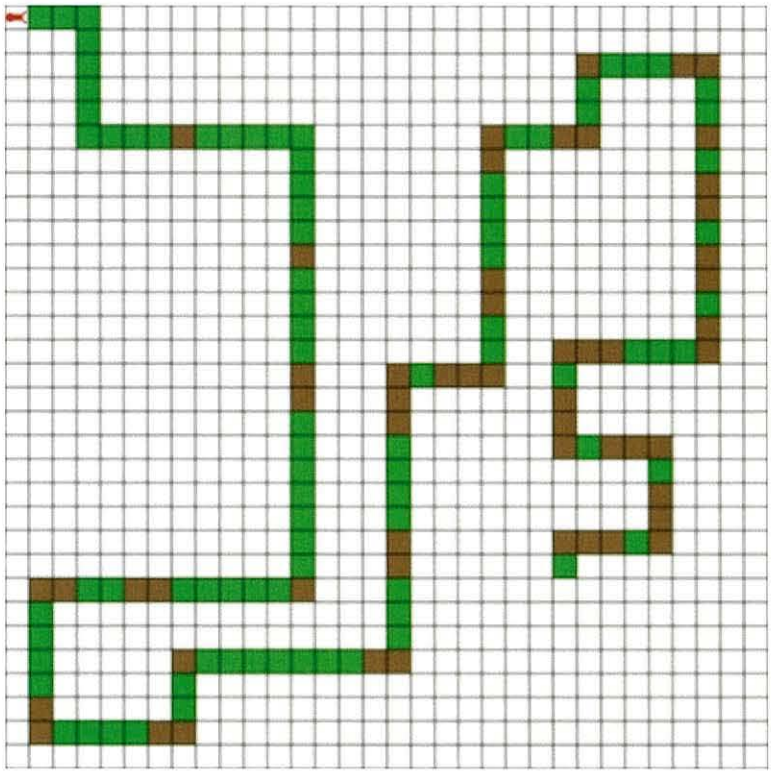


Figure 2.6: The Santa Fe Trail. Cited in Koza, 1992, p.55.

The objective of this problem is to find a computer program to control an artificial ant, so that it can find all 89 pieces of food located on the grid (green squares). The artificial ant operates in a square 32 by 32 toroidal grid in the plane and starts in the upper left cell of the grid (0, 0) facing east. The trail consists of 144 squares with 21 turns, and the 89 units of food (green squares) are distributed non-uniformly along it. It has the following irregularities (Koza 1992, p.54): single gaps, double gaps, single gaps at corners, double gaps at corners (short knight moves), and triple gaps at corners (long knight moves). The specifications of the problem are shown in Table 2.1.

Table 2.1: Santa Fe Trail problem specifications.

Objective	Find a computer program to control an artificial ant, so that it can find all 89 pieces of food located on the Santa Fe Trail.
Termination	Ant found all food units or maximum allowed steps reached.
Grid	32x32 Toroidal.
Trail	Consists of 89 Food Units, 55 Gaps, and 21 Turns.
Ant Start Position	Upper Left cell (0,0) Facing East.
Operations	*move*: Moves the ant forward in the direction it is currently facing. When the ant moves in a square, it eats the food, if there is any, in that square (thereby eliminating food from that square and erasing the trail). *turn right*: Turns the ant right by 90° (without moving the ant). *turn left*: Turns the ant left by 90° (without moving the ant). *food ahead*: Senses if there is food ahead. If yes, then returns true, otherwise it returns false.
Time (steps)	1 step for *move, turn right, turn left*, 0 steps for *food ahead.*
Maximum Steps	600

The Santa Fe Trail was designed by Christopher Langton (Koza 1992, p.54), and is a somewhat more difficult trail than the "John Muir Trail" originally used in the Artificial Ant problem by Jefferson, et al. (1992). Although there are more difficult trails than the Santa Fe Trail, such as the "Los Altos Hills" (Koza 1992, p.156), it is acknowledged as a challenging problem for evolutionary algorithms to tackle.

The problem has become quite popular as a benchmark in the Genetic Programming field and is still repeatedly used (Robilliard, et al., 2006) because of its interesting features. It has been shown by Langdon and Poli (1998a; 1998b) that Genetic Programming does not improve much on pure random search in the Santa Fe Trail because it contains multiple levels of deception. Langdon and Poli (1998a) argue that this problem may be indicative of real problem spaces.

Hugosson, Hemberg, Brabazon and O'Neill (2010) argue that the Santa Fe Trail problem can be considered a deceptive planning problem with many local optima. They mention that the Santa Fe Trail has the features often suggested in real program spaces – it is full of local optima and has many plateaus. Furthermore, they note that a limited GP schema analysis shows that it is deceptive at all levels and that there are no beneficial building blocks, leading genetic algorithms to choose between them randomly. These reasons make the Santa Fe Trail a challenging problem for evolutionary algorithms.

Los Altos Hills

The Los Altos Hills problem (Figure 2.7) was introduced by Koza (1992, p. 156). The objective of this problem is to find a computer program to control an artificial ant, so that it can find all 157 pieces of food (green squares) located on a 100x100 toroidal grid. The ant starts in the upper left cell of the grid (0, 0) facing east. The trail consists of 221 squares with 29 turns, and the 157 units of food (green squares) are distributed non-uniformly along it.

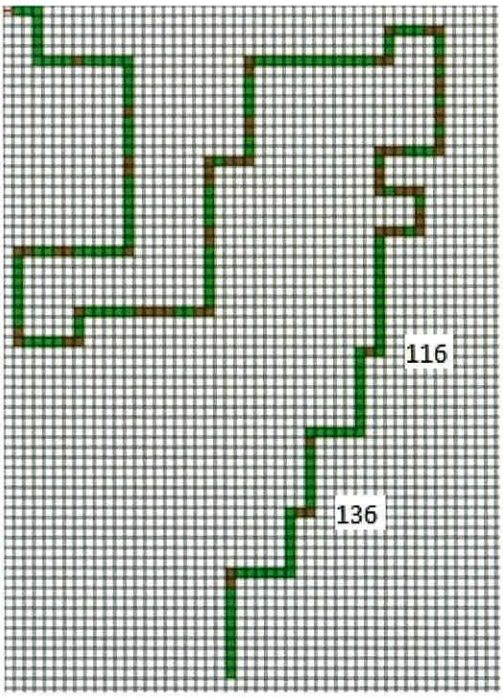


Figure 2.7: The Los Altos Hills trail (only the part of the grid with the trail is displayed here). Cited in Koza, 1992, p.157.

This problem has a larger grid and a larger and more difficult trail than the Santa Fe Trail problem. The Los Altos Hills trail begins with the same irregularities and in the same

order as the Santa Fe Trail. Namely, single gaps, double gaps, single gaps at corners, double gaps at corners, and triple gaps at corners. Then, it introduces two new kinds of irregularity which appear toward the end of the trail. The first and simpler requires a search of locations two steps to the left or two steps to the right of an existing piece of food. It appears for the first time at food pellet 116 (see Figure 2.7) of the trail. The second and more difficult irregularity requires moving one step ahead and then searching locations two steps to the left or two steps to the right of an existing piece of food. It appears for the first time at food pellet 136 (see Figure 2.7) of the trail.

2.5.5 Maze Searching Problems

A maze is a puzzle problem where the solver must find a route around a two-dimensional board from one point (start / entry) to another (end / exit) choosing between a series of complex and confusing pathways. The maze consists of fixed pathways and walls. The agent (traveller) can walk on a pathway but cannot cross or see behind a wall. The oldest known maze is the Cretan labyrinth.

Trying to develop a strategy for finding the goal state of a maze is a classic problem in Artificial Intelligence, for which there are various strategies, depending on the type of the maze, like for example the "hand on the wall" solution (Teahan 2010a, p.140). Blum and Kozen (1978) investigated the problem of visiting all path squares of a maze and they gave bounds on the power of systems of automata capable of mapping a maze, that is capable of deciding whether a path between two given squares in the maze exists, and they show that maze problems are easier to explore than graphs because of the availability of the orientation information.

The objective of the maze searching problem is to find a computer program to control an artificial traveller (agent), so that it can find the exit of the maze. The agent starts in the entry point of the maze facing the entry. The artificial traveller uses, like in the Artificial Ant problem, three primitive actions: *move*, *turn right*, and *turn left*. Each of these takes one time unit. In addition, the artificial traveller can use three sensing functions: *wall ahead*, *wall left*, and *wall right* (Sondahl, 2005), each of them requiring no time unit. These sensing functions look into the front, left or right square respectively, and return true if that square contains a wall or false if it is a path.

Maze searching problems are challenging for evolutionary algorithms because they contain local optima, obstacles often block the way toward the target and the walls present a series of obstacles parallel to each other that are very wide and difficult to find a way around. These characteristics of mazes increase the difficulty of finding an adequate evaluation function for an evolutionary algorithm.

Hampton Court Maze

The Hampton Court Maze is a simple connected maze of grid size 39 by 23 (Figure 2.8) which is a schematic representation of the Hampton Court Palace maze in the United Kingdom (Teahan 2010a, p.79). The entrance is at the middle bottom of the maze, and the exit is in the centre of the maze.

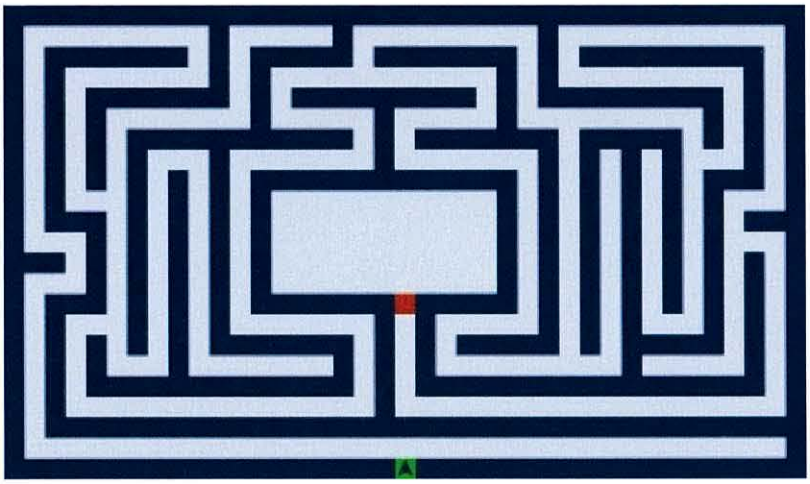


Figure 2.8: The Hampton Court Maze. Cited in Teahan 2010a, p.79.

As the Hampton Court maze is a simply connected maze, a human agent who adopts the behaviour of keeping the right or the left hand on the wall, will always be able to get to the centre of the maze which is the goal. Even though the maze can be solved with this simple strategy, Teahan (2010b, p. 128) states that the Hampton Court Maze is a challenging maze problem for search algorithms due to the presence of many local optima. For example, the square at the front of the entry point constitutes such a local optima.

Chevening House Maze

The Chevening House Maze (Figure 2.9) is a schematic representation of the Chevening House Maze in England (Teahan 2010a, p.82). It is a multiple-connected maze of grid size 47 by 47.

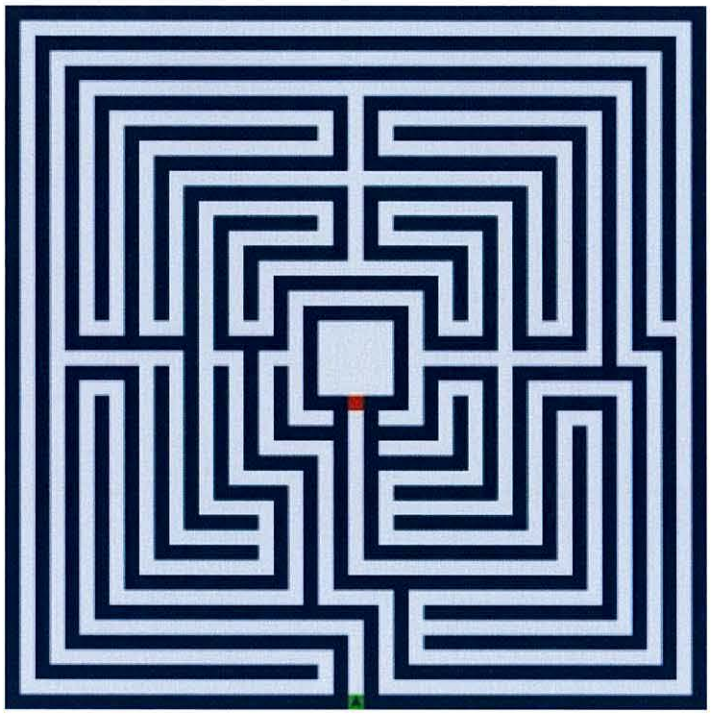


Figure 2.9: The Chevening House Maze. Cited in Teahan, 2010a, p. 83.

According to Teahan (2010a, p. 82), the Chevening House, built in the 1820s, was one of the first mazes that was multiple-connected. Namely, a maze that contains one or more "islands" that are isolated from each other, totally disconnected from the outer wall. An important difference between the Chevening House maze and the Hampton Court maze is that the former has been designed to thwart the "hand on the wall" behaviour for solving the maze (Teahan 2010a, p. 142).

2.6 Summary and Discussion

Natural evolution and genetics inspired the development of a new paradigm of problem solving, a family of population-based meta-heuristic search algorithms utilising the concepts of biological reproduction and evolution. The general idea behind this paradigm is to maintain a population of artefacts represented using a computer-based data structure,

reproduce these artefacts and evolve them directed by a fitness measure toward the goal of the task in hand in order to emerge a solution for the problem in question. The first most noticeable approaches were Evolutionary Programming, Evolution Strategies, and Genetic Algorithms, which have been unified under the umbrella term evolutionary algorithms and formed the foundation of the resultant field of evolutionary computation. A significant approach is Genetic Programming which directly evolves computer programs.

Research findings of the field revealed that there is not a unique configuration of an evolutionary algorithm which effectively solves any class of problems and that achieving a proper balance between exploration and exploitation is one of the most important factors in designing and configuring an evolutionary algorithm. Indeed, in order to achieve performance improvement, a match must exist between the structure of the algorithm and the structure of the problem. Another important aspect of an evolutionary algorithm, which affects its performance, is whether it is able to preserve diversity in the population in order to prevent the problem of premature convergence.

Even though, evolutionary algorithms have achieved remarkable results, research is underway trying to improve them, apply them to new areas (for example the emergence of autonomous agents behaviour, morphology, and design), and solve their issues. The most important of these issues are the lack of an effective modelling of the meta-learning based progress of biological evolution, the computational cost of these algorithms, and the lack of a unified view that would result in a common model and a minimum core and solid theoretical and practical background, where each approach would be expressed as a parameter or specialization.

A rather new development, a form of Genetic Programming named Grammatical Evolution, is a type of automatic programming that resolves the Genetic Programming and grammar-based algorithms issue of generation and preservation of valid programs (knows as the "closure" problem), due to the use of variable length binary string genomes, a unique genotype-to-phenotype mapping formula, and a genotype wrapping mechanism. The ability of this algorithm to create from binary strings valid programs in any programming language, which can be expressed with a BNF grammar definition, lead to its application on a variety of benchmark and real word problem as well as to further research and investigation. Variations of the original algorithm have been already implemented which try to tackle its issues such as destructive crossovers, genotype

bloating, dependency problems, and low locality of genotype-to-phenotype mapping. A promising research direction is the utilisation of modularity for which the past research in the area of Genetic Programming shows that increases the effectiveness and efficiency of an evolutionary run.

Grammatical Evolution has started to gain popularity as evidence by an increasing number of publications and with the appearance of publicly available implementations in various programming languages such as C++, Java, Python, and Ruby.

Finally, evolutionary algorithms are benchmarked in a variety of standard problems. Most of them could be considered as toy-problems such as symbolic regression and symbolic integration. But there are also problems of interest in such areas as evolutionary robotics which involve the evolution of the behaviour of agents in a given environment such as the artificial ant problem and the maze searching problem. Even though these problems could be also considered as toy-problems, they reveal some interesting attributes of real world problems, such as many deceptive local optima and large search spaces, which makes them challenging and difficult for evolutionary algorithms to solve. An interesting conclusion that is revealed from the survey that is conducted in this chapter is that previous Grammatical Evolution literature has not investigated applications to maze searching problems and to the Los Altos Hills problem, an instance of the artificial ant problem which is more challenging than Santa Fe Trail.

Chapter 3

Java Grammatical Evolution

3.1 Introduction

The purpose of this chapter is to describe the Java Grammatical Evolution (jGE) Library, to demonstrate its use in proof-of-concept experiments, and to compare it with other widely known GE implementations. The main idea behind jGE, besides the development of a Grammatical Evolution implementation, is the creation of a generic framework for evolutionary algorithms which can be extended with the incorporation of standard or new evolutionary algorithms (sequential and parallel), and the addition of standard or new benchmarking problems. Part of the work in this chapter has been published as an extended abstract in Georgiou and Teahan (2006a; 2006b; 2008). However, the work here provides greater detail, and includes a full description of the experimental results.

The jGE Library was the first publicly available (Georgiou, 2006) and published (Georgiou and Teahan, 2006a) implementation of Grammatical Evolution for the Java programming language (Java SE versions 5 and 6), and was developed for the facilitation of the work described in this text. Today, it is the core component of the jGE Project at the School of Computer Science of Bangor University (Artificial Intelligence and Intelligent Agents Research Group). The goal of this project is the implementation of an open Evolutionary Algorithms (EA) framework which will facilitate further research into evolutionary algorithms (and especially Grammatical Evolution). Grammatical Evolution was chosen as the main evolutionary algorithm of the jGE Project mainly because it facilitates, due the use of a BNF Grammar, the evolution of arbitrary structures and programming languages (Georgiou and Teahan, 2006b).

The main objectives of the jGE Project are as follows:

- to provide an open and extendable framework for the experimentation with evolutionary algorithms;
- to create an agent-oriented evolutionary system (using an agent-based framework);

- to bootstrap further research on the application of natural and molecular biology principles like for example population thinking (Mayr 2002, p.81); and
- to facilitate experimentation in the evolution of populations of agents of third-party open source and free Java projects. For example, evolving competent Robocode (Nelson, 2001) robots capable to beat human designed robots.

Java was chosen as the implementation language for the jGE Library, mainly to fit in with other artificial intelligence projects being developed at Bangor. However, Ghanea-Hercock (2003, p.15) also lists several advantages of using Java for the development of Evolutionary Algorithms applications, like automatic memory management, pure object-oriented design, high-level data constructs (e.g. dynamically resizable arrays); platform independent code; and the availability of several complete EA libraries for EA systems. On the other hand, Ghanea-Hercock (2003, pp.15-16) mentions that the main price we have to pay in using Java is the significant increase in execution time of interpreted Java programs compared with compiled languages like C and C++. But he adds that the recent work of Sun Microsystems Inc. and other companies has resulted in "Just in Time" compilers which significantly improve the execution speed of the Java programs. Also, he mentions that future developments in computer languages may lead to better alternatives to Java with for example improved speed. Such an example is the release of C# from Microsoft (Ghanea-Hercock 2003, p.17). Of course, in spite of the last argument of Ghanea-Hercock, there is a huge dispute of whether Java is slower or faster than C# (or even than C and C++). And to put it in the correct context: whether a particular implementation of the JVM (Java Virtual Machine) is slower or faster than one of the CLR (Common Language Runtime) of Microsoft.

Another advantage of using Java, a purely object-oriented programming (OOP) language, against languages of other programming paradigms like procedural or functional (e.g. C, Cobol, Basic, Common Lisp, Scheme), are according to Alba and Troya (2001) the following: a) Quick prototyping of the evolutionary algorithm with quick redefinition of the classes involved and consequently easier identification of the best algorithm; b) object-oriented programming facilitates a high level approach of experimentation; c) software reuse; and d) the development of a common architecture of classes for an evolutionary system.

Furthermore, the intention for the incorporation of parallel evolutionary algorithms in a future version of the jGE Library, gives to Java another one advantage. Alba and Troya (2001) argue that Java multi-platform parallelism and heterogeneity at virtually "zero cost" is a very appealing feature for future research with parallel algorithms, and they provide results which show the efficiency and flexibility of the utilization of object-oriented programming in the domain of Parallel Evolutionary Algorithms (PEA). Namely, they give some hints for designing PEAs with OOP regarding the stochastic decisions (ensuring a single seed), the population implementation (using dynamic lists), the computer memory saving (reducing the memory spent in the basic classes), and the communication via sockets (the explicit closing of useless sockets). They conclude that OOP allows quick PEA prototyping, integration of new techniques within the PEA and easy cooperation with other techniques in parallel (all of them without reducing the efficiency of the resulting PEA).

3.2 Overview of the jGE Library

This and the three next sections describe the overall architecture of jGE, overview its components & how they are organised in to packages, and provide further detail of one of its prominent components, the Genetic Operations. The complete API documentation of the jGE Library can be found in Georgiou (2006) and in the accompanying CD. The API documentation provides a detailed description of all the classes of the library and their methods.

The main idea behind the development of the jGE Library – it can be downloaded at Georgiou (2006) – is to create a framework for evolutionary algorithms which can be extended to any specific implementation such as Genetic Algorithms, Genetic Programming, Grammatical Evolution. This means that instead of using a mapper-centric approach like libGE (Nicolau, 2005), jGE uses an EA-oriented approach. Namely, instead of being just the implementation of the mapping mechanism between the *Search Engine* and the *Evaluation Engine* as for libGE, it provides libraries for both of these components. Consequently, someone using jGE is able to specify the core strategy of the evolutionary process by selecting and/or setting the following (see Figure 3.1):

- the desired implementations of the genetic operators (crossover, mutation, duplication, pruning, parents selection mechanism, offspring selection mechanism, standard evolutionary algorithms templates, etc.);
- the genotype to phenotype mapping mechanism;
- the evaluation mechanism;
- the initial population; and
- the initial environment (although currently not yet implemented).

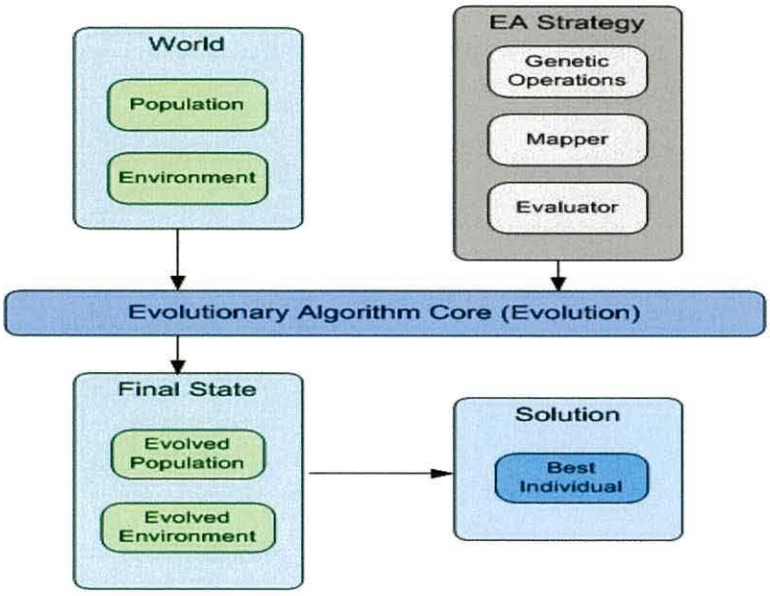


Figure 3.1: The jGE architecture.

Even though jGE is focused on the implementation of the Grammatical Evolution system, it contains all the necessary functionality for the execution and/or construction of other evolutionary algorithms as well – currently, as well as GE, two other EC algorithms are implemented in the library: Standard GA (Generational) and Steady-State GA. Namely, jGE decomposes and implements services which are required by evolutionary algorithms and provides functionalities for the ad-hoc implementation of evolutionary based systems.

In the current version, the library is concentrated on Grammatical Evolution, but it can also be used by any other Java system or program for the creation of evolutionary algorithms as well as for other functionalities such as the parsing and representation of BNF Grammar definitions, the compilation and execution of Java programs, and the generation of random numbers in specific ranges.

3.3 Components of jGE

The diagram in Figure 3.2 shows the main components of the jGE Library and the diagram in Figure 3.3 shows the classes of its core components.

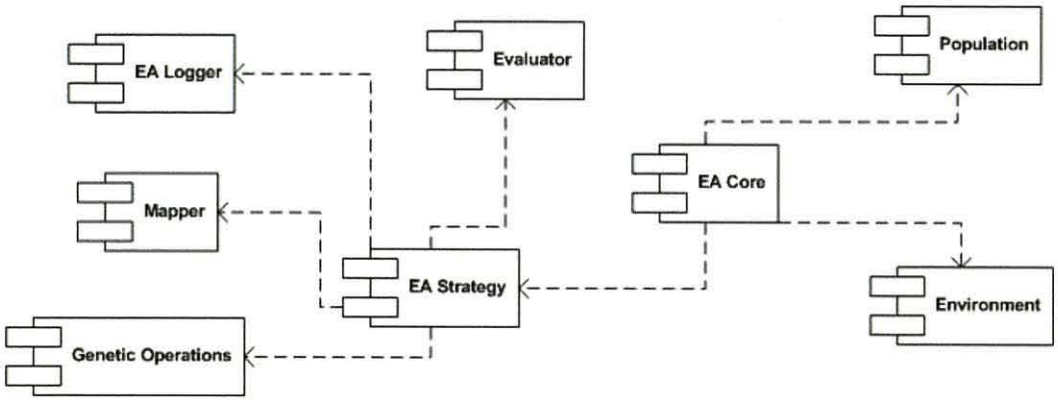


Figure 3.2: Component diagram of the jGE Library.

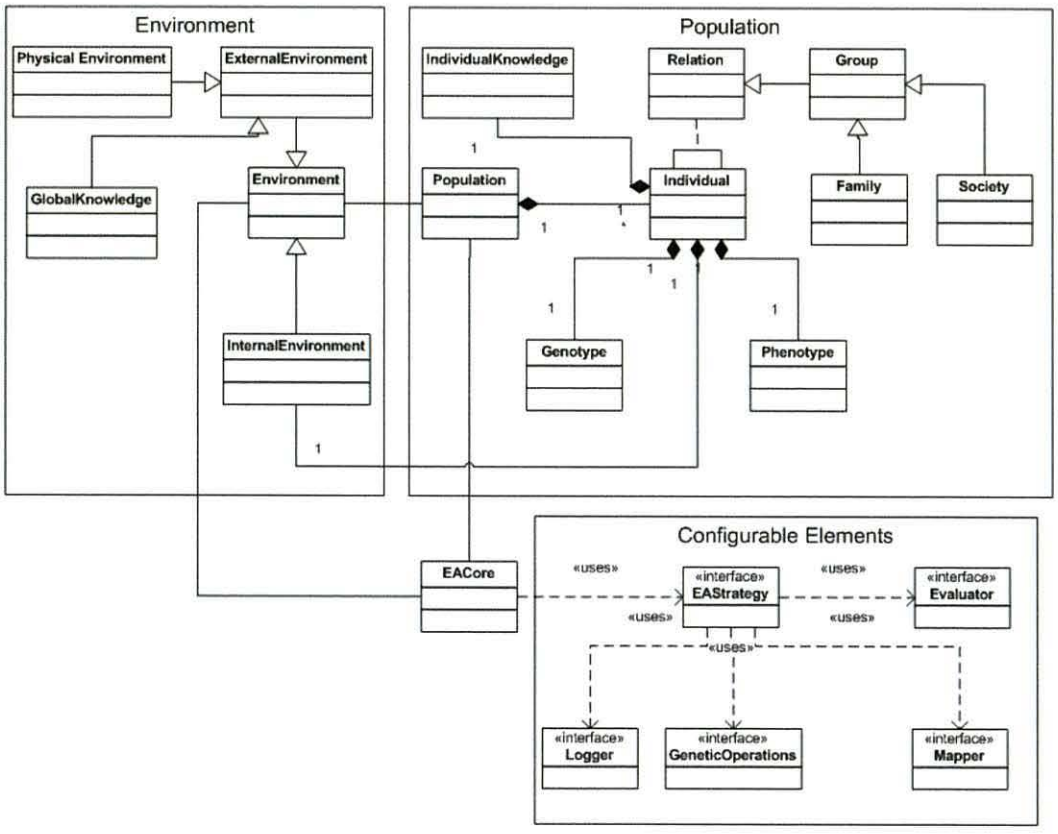


Figure 3.3: Class diagram of the core components of the jGE Library.

EA Core

This component is the main entry to the platform and loads the necessary files and classes. Namely, it loads the *Population* component, the *Environment* component, and the *EA Strategy* to be executed. Also, it is responsible for checking the supported features and configuration possibilities of the *EA Strategy* component.

EA Strategy

The *EA Strategy* component defines the evolutionary process to be executed and the specification of the problem in question. It is responsible for selecting the appropriate implementations of each one of the configurable elements: *Genetic Operations*, *Mapper*, *Evaluator*, and *Logger*. The evolutionary strategy and the problem are currently specified with Java code but in a future version they could be determined by a configurable external XML file which will be loaded by the core mechanism of the jGE framework (*EA Core*). The core mechanism will be then responsible for allocating and executing the appropriate actions and directives, and to produce the final results.

Population

It contains a collection of classes which abstract a real world population. Namely, it provides classes for the representation of the individuals (their genotypes and phenotypes). Also it can be extended in future versions of the library with classes representing relations between the individuals, groups of individuals, and more advanced concepts such as individual's knowledge.

Environment

This component contains classes which abstract a real world environment. Its purpose is to allow the researcher to specify an environment in which the population is situated, and to influence the creation of new generations as well as the phenotype of the individuals, their growth process, and finally their own genotypes before they reproduce new offspring. This component is not yet completed in the current version of the jGE Library.

Genetic Operations

It contains an extendable collection of implementations of variations of the genetic operations. Namely, it provides templates and various implementations of evolutionary algorithms (generational GA, steady-stage GA), reproductive operators (crossover, mutation, duplication, pruning), selection mechanisms (parents selection, offspring selection), and more. This component is described in detail in section 3.5.

Mapper

The *Mapper* component is responsible for the genotype-to-phenotype mapping of an individual. It provides an interface which accepts as an argument an individual's genotype and returns its corresponding phenotype. An implementation of any mapping process can be added and used in jGE as long as it satisfies the required interface. Currently, two mapping processes are implemented and supported: No-Mapping (e.g. Genetic Algorithm) and BNF-Based Mapping (Grammatical Evolution). Any specific implementation of the mapping process will be executed by the *EA Strategy* component.

Evaluator

The *Evaluator* is used by the *EA Strategy* to assign a fitness value to the phenotype of an individual. It defines a standard interface and any implementation of a problem specification must implement this interface. In the current version of jGE, two problem specifications (and their corresponding evaluators) are available: Hamming distance and symbolic regression. The *Evaluator* implementation is the only component of the system that has to be created for any new category of problems which will be tackled by the jGE Library.

EA Logger

This keeps track of the evolutionary process and stores the data for monitoring of the evolution of the individuals and for later use; that is, the creation of statistical results.

Regarding the communication between the different components, the first thought was that the *Mapper*, *Evaluator*, *Genetic Operations*, and *Logger* components should not use directly the *Population* and *Environment* components. They would get and send back data

from and to the *EA Strategy* component using the *String* data type. This would increase the modularity and flexibility of the system but at the same time would decrease opportunities for future extensions. For example, in a future extension the genetic operations could need to take into account the environment and global/individual knowledge or the relations between the individuals. For this reason, the design decision has been taken that the *Population* and *Environment* components should be used directly by every other component of the system.

3.4 The jGE Packages

The components and classes of the jGE Library (see Figure 3.2 and Figure 3.3) are organised in the packages described below. A diagram depicting the main classes of each package is shown in Figure 3.4. As already noted, the detailed description of each class and its methods can be found in the API Documentation of the library (Georgiou, 2006).

3.4.1 Package: bangor.aiia.jge.core

The package *core* contains the main classes of the jGE library. These classes implement the *EA Core* component, the *EA Strategy* component, the Grammatical Evolution algorithm, and some proof of concept experiments. Also, the interfaces of this package define the required functionality which must be provided by specific implementations of the *Evaluator* and *Mapper* components of the system. Indeed, it provides two implementations of the later (no mapping and GE mapping). See Appendix F for the class diagram of the *Mapper* component of this package.

3.4.2 Package: bangor.aiia.jge.population

This package contains the classes which represent the population of an evolutionary algorithm. A *Population* is a collection of *Individuals* and each *Individual* has a *Genotype* and a *Phenotype*. See Appendix F for the class diagram of this package.

3.4.3 Package: bangor.aiia.jge.evolution

The package *evolution* contains implementations of evolutionary algorithms (e.g. GA) and classes which decompose the main operations of such algorithms (e.g. crossover,

mutation, selection, etc.). These operations are implemented as static methods and each class provides a collection of different variations. The classes include: crossover; duplication; genesis; mutation; pruning; selection. For example, the class *Crossover* currently provides a standard one-point crossover operator for both fixed and variable length genomes, and the class *Selection* provides two standard selection mechanisms: rank selection and roulette wheel selection. Alternative implementations of the provided operations, such as two-point and uniform crossover variations, can be added as static methods to the corresponding Java classes.

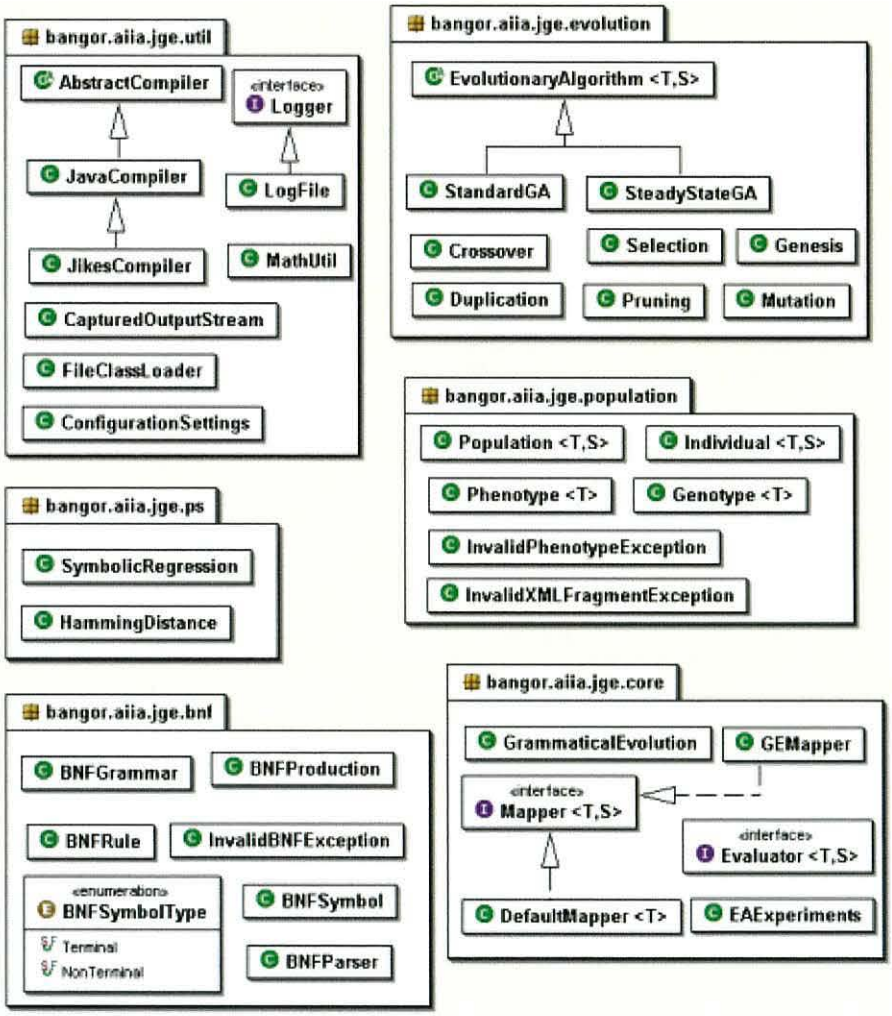


Figure 3.4: jGE Library packages.

3.4.4 Package: bangor.aiia.jge.ps

The *ps* (problem specification) package contains specific implementations of the *Evaluator* component, namely problem specifications which can be used for the

evaluation of the individuals during an EA run. Each time a new problem is examined, a new problem specification class has to be created which assigns a fitness value to an *Individual* of the *Population*. The classes *HammingDistance* and *SymbolicRegression* implement Hamming distance and symbolic regression problems respectively. See Appendix F for the class diagram of this package.

3.4.5 Package: bangor.aiia.jge.bnf

This package contains all the necessary classes for the loading and validation of a BNF grammar definition and its representation as Java objects. Currently, only the classical BNF dialect is supported which conforms to the following rules: the terminal symbols are written naturally; the non-terminals symbols are enclosed in angle brackets, i.e. `<symbol>`; the definition symbol in rules is "`::=`"; there is no rule statement terminator (this means that each rule terminates when the next one begins). Further details and samples can be found in the jGE API documentation (class *bangor.aiia.jge.bnf.BNFParser*). Also, see Appendix F for the class diagram of this package.

3.4.6 Package: bangor.aiia.jge.util

The package *util* contains the *EA Logger* component and some helper classes (utilities). The classes of this package implement common services used by other components and classes of the jGE Library, like logging, compilation and execution of Java code, stochastic functions, and more.

3.5 Genetic Operations Component

One of the most useful and important components of the jGE Library is the *Genetic Operations* component. Its classes implement standard evolutionary algorithms such as GA, and various types and variations of standard genetic operators as static methods. In this way, ad-hoc implementations of evolutionary algorithms can easily access the various genetic operations and use them in different combinations. Currently the following operators are implemented:

- *Genesis*: random creation of an initial pool of binary string genotypes; and random creation of an initial population of individuals.

- *Selection*: roulette wheel selection; rank selection; N best and M worst selection.
- *Crossover*: standard one-point crossover for fixed-length genotypes; standard one-point crossover for variable-length genotypes.
- *Mutation*: standard one-point mutation.
- *Duplication*: standard Grammatical Evolution duplication.
- *Pruning*: standard Grammatical Evolution pruning.

The abstract class *EvolutionaryAlgorithm* defines common properties and behaviours for evolutionary algorithms like Genetic Algorithms, Genetic Programming, and Grammatical Evolution. An evolutionary algorithm simulates the biological process of evolution. The evolution unit of this process is the population as Darwinism argues (Mayr 2002, p.9). The basic strategy of an Evolutionary Algorithm is listed in Listing 3.1.

```
Set current population P = N individuals
For Generation = 1 to MaxGenerations
  Competition: Evaluate the individuals of P
  Selection: Select from P the individuals to mate
  Variation: Apply Crossover, Mutation, etc. to the selected
    individuals
  Reproduction: Create the new population P' and set P = P'
End For
```

Listing 3.1: Evolutionary algorithm basic strategy.

The subclasses of the *EvolutionaryAlgorithm class* must implement the concrete steps of the above strategy in order to provide specific versions of evolutionary algorithms.

Further, two evolutionary algorithms have been implemented: the Standard Genetic Algorithm (Generational GA) and a version of a Steady-State Genetic Algorithm. For the former, the following process as shown in Listing 3.2.

```
Set current population P = N individuals
Perform fitness evaluation of the individuals in P
While (solution not found and MaxGenerations not exceeded)
  Create a new empty population, P'
  Repeat until P' is full
    Select two individuals from P to mate using Roulette Wheel
      Selection
    Produce two offspring using standard one-point crossover
```

```
      with probability Pc
    Perform Point Mutation with probability Pm on the two
      offspring
    Perform Duplication with probability Pd on the two offspring
    Perform Pruning with probability Pp on the two offspring
    Add the two offspring into P'
  End Repeat
  Replace P with P'
  Perform fitness evaluation of individuals in P
End While
Return the best individual, S (the solution) in current
  population, P
```

Listing 3.2: Standard genetic algorithm.

For the Steady-State Genetic Algorithm (SSGA), the main idea is that a portion of the population P survives in the new population P' and that only the worst individuals are replaced. Namely, a few good individuals will mate and their offspring will replace the worst individuals. The rest of the population will survive. The SSGA process implemented in jGE is shown in Listing 3.3.

```
Set current population P = N individuals
Set G = the generation gap
Perform fitness evaluation of the individuals in P
While (solution not found and max generations not exceeded)
  Create a new empty population, P'
  Repeat until (new offspring = G × N)
    Select two individuals from P to mate using Roulette Wheel
      Selection
    Produce two offspring using standard one-point crossover
      with probability Pc
    Perform Point Mutation with probability Pm on the two
      offspring
    Perform Duplication with probability Pd on the two offspring
    Perform Pruning with probability Pp on the two offspring
    Add the two offspring into P'
  End Repeat
  Add the best (N - G × N) individuals of P into P'.
  Replace P with P'
  Perform fitness evaluation of individuals in P
End While
Return the best individual, S (the solution) in current
  population, P
```

Listing 3.3: Steady-state genetic algorithm.

The portion of the population P that will be replaced in P' is known as the *generation gap* and is a fraction in the interval (0, 1). The default implementation of SSGA uses a fraction, $G = 2/N$ (where N the size of the population). Namely, two individuals will mate and their offspring will replace the two worst individuals. In general, the number of the

individuals which will be replaced in each generation is $G \times N$. In case where $(G \times N)$ is not an even integer, then the larger even integer less than $(G \times N)$ and larger than 0 will be used.

The *GrammaticalEvolution* class of the *EA Core* component uses the previously mentioned classes of the *Genetic Operators* component to implement the default version of the Grammatical Evolution algorithm, with a minor exception regarding the steady state replacement mechanism as mentioned. The default implementation, as described by O'Neill, Ryan, Keijzer and Cattolico (2001), uses a *Steady-State* replacement mechanism such that two parents produce two children, the best of which replace the worst individual in the population only if the child has a greater fitness than the individual to be replaced. The jGE implementation uses a slightly different replacement mechanism which is described above in the SSGA process (Listing 3.3). Also, there is the option to use a *Generational* replacement mechanism like in standard GA.

Regarding the configuration of a Grammatical Evolution run, O'Neill and Ryan (2001; 2003) use the following: a typical wrapping threshold is 10; the size of the codon is 8-bits; and typical probabilities are: crossover–0.9; mutation–0.01; duplication–0.01; pruning–0.01. This configuration is the default of the *GrammaticalEvolution* class. Further, this implementation uses the following default values: max. generations: 10; searching mechanism: Steady-State GA; Generational Gap of the Steady-State GA $2/N$ (N is the population size).

The next section describes some proof-of-concept experiments performed with the jGE Library.

3.6 jGE Demonstration Experiments

Three different demonstration experiments with jGE were performed – Hamming distance, symbolic regression and trigonometric identity (see forthcoming subsections). These are proof-of-concept experiments of the Grammatical Evolution implementation in the jGE Library. For the first problem, two further evolutionary algorithms were tried for comparison – Standard (Generational) GA, and Steady State GA. The second and third problems are based on the experiments which have been performed by O'Neill and Ryan (1998; 2001; 2003, pp.49-52). The objective of these experiments is to demonstrate the

applicability and effectiveness of the jGE Library for the execution of evolutionary computation experiments.

Before the results are described, however, the next subsection provides a brief discussion on some of the Java issues encountered during the experiments, and this is followed by a subsection providing some sample Java source code to illustrate the ease with which these experiments were set up using the jGE Library.

3.6.1 Java Issues

The first version of the *Evaluator* component used the *JavaCompiler* class to evaluate the Java programs (phenotypes). This compiles (using the Sun's *javac.exe* compiler), and executes (using the Sun's *java.exe* runtime), once in each generation of a run, the dynamically created Java source code which are the phenotypes of all the individuals of the population. This is an extremely time consuming task and for problems such as symbolic regression, this is the most important factor which affects the execution speed. In each symbolic regression experiment, the compilation-execution takes place once when a new run starts (for the creation of the initial population) and once in each generation (during the evaluation of the individuals of the population).

Although the time complexity with respect to the compilation-execution of Java code is linear (*O(N)* where *N* is the number of generations of a run), it is a time consuming task which can significantly degrade overall performance. Moreover, other problems will have a higher rate of growth of execution time if they need to frequently use the source code compilation and Java bytecode execution tasks.

For the above reasons, alternative methods for the compilation and execution of Java code were investigated. The experimental evidence (see Georgiou, 2006) leads to the conclusion that a much better solution than using *javac.exe* and *java.exe* is the following setup: a) Use of the Jikes compiler for the compilation of the Java source code (IBM Corporation, 2004); b) Utilization of the Dynamic Class Loading and Introspection features of the Java Virtual Machine (*ClassLoader* class, and the *Reflection* API).

Jikes is an open source Java compiler written in the C++ language and translates Java source files into the bytecode instructions set and binary format defined in the Java

Virtual Machine Specification. Jikes has the following advantages as noted in the Jikes official web site: open source; strictly Java compatible; high performance; dependency analysis; constructive assistance (IBM Corporation, 2004).

The Java *ClassLoader* is an important component of the Java Virtual Machine which is responsible for finding and loading classes at runtime. It loads classes on demand into memory during the execution of a Java program. Furthermore, it is written in the Java language and can be extended in order to load Java classes from every possible source (local or network file system, network resources, etc.). Using both the *ClassLoader* and the *Reflection* API, it is possible to perform the loading of Java bytecode and its execution from inside of any Java program using the same instance (process) of the JVM. In the current version, the jGE Library provides the option of using either the Sun's JVM or the IBM's Jikes compiler in conjunction with Dynamic Class Loading and Introspection for the compilation and execution of Java code.

The experiments described below are the first experiments with jGE using real data based on the suggested configurations provided by O'Neill and Ryan (2003, p.50). Because this was the first time a large amount of data was used by jGE (e.g. populations of 500 individuals, sample of 50 data points etc.), an unexpected problem arose. During the evaluation of a Grammatical Evolution run on a symbolic regression problem, the Java compilers (both Sun JDK 1.5 and IBM Jikes) threw an error during the compilation of the produced Java class which was responsible for calculating the raw fitness of all the individuals of a population. The error message in Sun JDK 1.5 was the following: "Code too large". The reason for this error was tracked down to an undocumented limitation of the Java compiler which cannot compile a method with bytecode size larger than 64Kb.

This problem forced the re-factoring of the *SymbolicRegression* class in order for the compilation of the class which runs and evaluates the Java code (phenotype) of the individuals to be possible. The whole code which was placed in the main method of a temporary created class, had been broken into many smaller methods instead (one for each individual of the population).

3.6.2 Sample Java Source using jGE

This section provides a sample of the Java source code used for these experiments. The source code used for all three problems is essentially the same – except for a small amount of variation to specify the problem itself and the evolutionary algorithm used. Listing 3.4 shows the Java source code for the Hamming distance problem experiment. The line labelled by (1) in the listing provides the problem specification (this will vary for the three types of problems). Lines labelled by (3) set the parameters to be used by the evolutionary algorithm. The line labelled by (4) executes the algorithm and returns the solution found.

```
   // This method shows the use of jGE in a Hamming Distance problem.
   // @return The solution of the Hamming Distance experiment.
   public static Individual<String, String> hdExperiment() {
      Individual<String, String> solution = null;
      String target = "11100011100010101010101010101010";
      LogFile log = null;

(1)   HammingDistance hd = new HammingDistance(target);
(2)   // Insert EA specification here (see Listing 3.5)
(3)   ea.setCrossoverRate(0.9);
(3)   ea.setMutationRate(0.01);
(3)   ea.setDuplicationRate(0.01);
(3)   ea.setPruningRate(0.01);
(3)   ea.setMaxGenerations(100);
(3)   ea.setLogger(log);
(4)   solution = ea.run();

      // Also the following information can be retrieved:
      // Number of Generations created = ea.lastRunGenerations();
      // Solution's Fitness Value = solution.rawFitness();
      // Solution's phenotype = solution.getPhenotype().value().

      return solution;
   }
```

Listing 3.4: Java source code for the Hamming distance problem experiment.

Listing 3.5 shows the alterations needed to the source code in Listing 3.4 to configure for different evolutionary algorithms. That is, line (2) should be replaced by the code shown in Listing 3.5 depending on the algorithm that is chosen. Additionally, an extra line needs to be inserted before (3) but only for the Steady State Genetic Algorithm. Any of the variations of the method shown in Listing 3.4 performs a simple run of the corresponding experiment. An external application is needed to call the method using a loop in order to execute a full experiment with many evolutionary runs.

```
Standard GA:
Replace line (2) in Listing 3.4 with:
StandardGA ea = new StandardGA(50, 1, 30, 30, hd);

Steady-State GA:
Replace line (2) Listing 3.4 with:
SteadyStateGA ea = new SteadyStateGA(50, 1, 30, 30, hd);
Insert before line (3) above:
ea.setFixedSizeGenome(true);

Grammatical Evolution:
Replace line (2) Listing 3.4 with:
BNFGrammar bnf = new BNFGrammar("BinaryGrammar.bnf");
GrammaticalEvolution ea = new GrammaticalEvolution(
                              bnf, hd, 50, 8, 20, 40);
```

Listing 3.5: Source code alterations to Listing 3.4 required for the different evolutionary algorithms in the Hamming distance problem experiments.

The next subsections describe the experimental results for the three problems investigated. Each subsection provides a description of the problem, the configuration of the experiment, the BNF grammar used by the Grammatical Evolution system, and the experimental results.

3.6.3 Hamming Distance Experiments

The Hamming distance problem involves the finding of a given binary string. The target string was: 11100011100010101010101010101010. For this problem, Grammatical Evolution, Standard GA, and Steady-State GA were compared. Listing 3.6 shows the BNF grammar definition used in this experiment. The tableau of Table 3.1 – with a style similar to O'Neill and Ryan (2001; 2003) of summarising information – shows the configuration of the experiment.

```
<phenotype> ::= <binary><binary><binary><binary>
                <binary><binary><binary><binary>
                <binary><binary><binary><binary>
                <binary><binary><binary><binary>
                <binary><binary><binary><binary>
                <binary><binary><binary><binary>
                <binary><binary><binary><binary>
                <binary><binary>
<binary>    ::= 0|1
```

Listing 3.6: BNF grammar used for the Hamming distance problem.

Table 3.1: Hamming Distance GE Tableau.

Objective:	Find the target binary string.
Terminal Operands:	*0* and *1*.
Terminal Operators:	None.
Fitness cases:	The target string.
Raw Fitness:	*Target String Length - Hamming Distance*.
Standardised Fitness:	*Target String Length - Raw Fitness*.
Wrapper:	None.
Parameters:	Population Size (M) = 50, Maximum Generations (G) = 100, Prob. Mutation (P_m) = 0.01, Prob. Crossover (P_c) = 0.9, Prob. Duplication (P_d) = 0.01, Prob. Pruning (P_p) = 0.01, Steady State GA with Generation Gap (G) = 0.9, Roulette-Wheel Selection, Codon Size = 8.

In each experiment, 100 evolutionary runs were performed. Table 3.2 depicts the success rates of these experiments (percentages of solutions found in each experiment). The cumulative frequency measure of success over 100 runs for the three algorithms can be seen in Figure 3.5.

Table 3.2: Results for the Hamming distance problem.

	Standard GA	Steady-State GA	Grammatical Evolution
Runs	100	100	100
Success Rate	10%	100%	44%

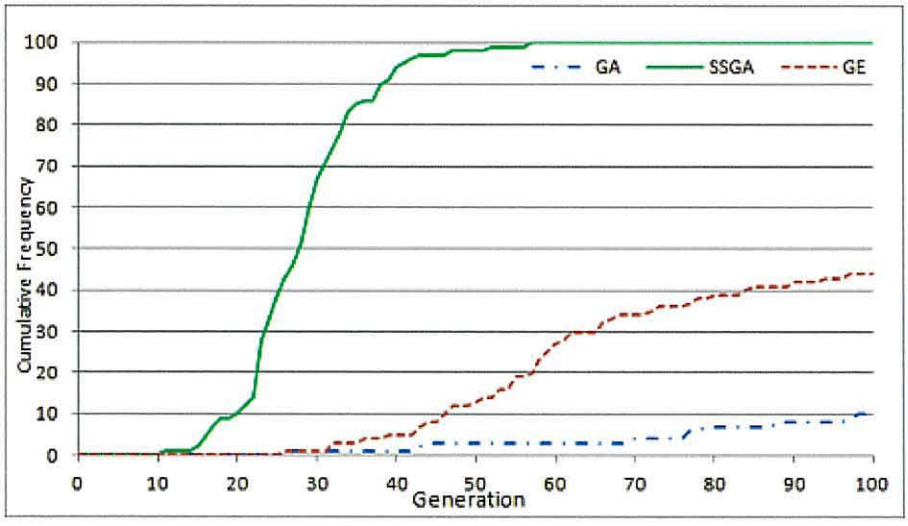


Figure 3.5: Hamming distance results graph.

The figures show that Grammatical Evolution (or more precisely, the jGE implementation of it) outperforms the Standard Genetic Algorithm (generational) achieving a higher success rate (44% against 10%) but a lower compared to the Steady-State Genetic Algorithm which achieved the absolute success rate (100%).

3.6.4 Symbolic Regression Experiments

Symbolic regression problems involve finding some mathematical expression in symbolic form that matches a given set of input and output pairs. The particular function experimented with was the following: $f(x) = x^4 + x^3 + x^2 + x$. Two different BNF grammars were tried (Listing 3.7 and Listing 3.8) using the configuration shown in Table 3.3.

```
<expr>    ::= <expr> <op> <expr> |
              ( <expr> <op> <expr> ) |
              <pre-op> ( <expr> ) |
              <var>
<op>      ::= + | - | / | *
<pre-op>  ::= Math.sin | Math.cos | Math.log
<var>     ::= x | 1.0
```

Listing 3.7: BNF grammar (A) used for the symbolic regression problem.

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

Listing 3.8: BNF grammar (B) used for the symbolic regression problem.

Table 3.3: Symbolic regression GE tableau.

Objective:	Find a function of one independent variable and one dependent variable, in symbolic form that fits a given sample of 20 (x_i, y_i) data points, where the target function is the quadratic polynomial $x^4 + x^3 + x^2 + x$.
Terminal Operands:	x (the independent variable), the constant *1.0*.
Terminal Operators:	The binary operators +, -, /, and *. The unary operators *Math.sin*, *Math.cos*, and *Math.log*.
Fitness cases:	The given sample of the pairs (x_i, y_i) of 20 data points in the interval [-1, +1]. The input data points (x_i) are randomly created and their corresponding output points (y_i) are automatically created by the expression $x^4 + x^3 + x^2 + x$.
Raw Fitness:	The sum of the absolute values of errors taken over the fitness

	cases (x_i, y_i). A lower raw fitness value indicates better individuals.
Standardised Fitness:	Same as raw fitness.
Adjusted Fitness:	$1 / (1 + Fs(i))$ where Fs is the Standardised Fitness of i. The fitness value varies from 0 to 1. Invalid individuals are assigned a zero (0) adjusted fitness value.
Hits:	The number of fitness cases for which the Adjusted Fitness is greater than or equal to 0.999.
Wrapper:	Standard productions to generate a Java class with a main() method which prints the fitness values in the standard output.
Parameters:	Population Size (M) = 500, Maximum Generations (G) = 50, Prob. Mutation (P_m) = 0.01, Prob. Crossover (P_c) = 0.9, Prob. Duplication (P_d) = 0.01, Prob. Pruning (P_p) = 0.01, Steady State GA with Generation Gap (G) = 0.9, Roulette-Wheel Selection, Codon Size = 8.

The experimental results, over a series of 100 evolutionary runs, show that the BNF grammar (B) is a much better grammar to use for tackling this problem as evidenced by the success rate in Table 3.5 being 68% as compared to 5% in Table 3.4 when the BNF grammar (A) is used.

Table 3.4: Results for symbolic regression using BNF grammar (A).

	GE using grammar (A)
Evolutionary Runs	100
Highest Fitness Value	1.0
Success Rate	5%

Table 3.5: Results for symbolic regression using BNF grammar (B).

	GE using grammar (B)
Evolutionary Runs	100
Highest Fitness Value	1.0
Success Rate	68%

The Figure 3.6 shows the cumulative frequency measure of success over 100 runs for Grammatical Evolution using grammar (A) and grammar (B).

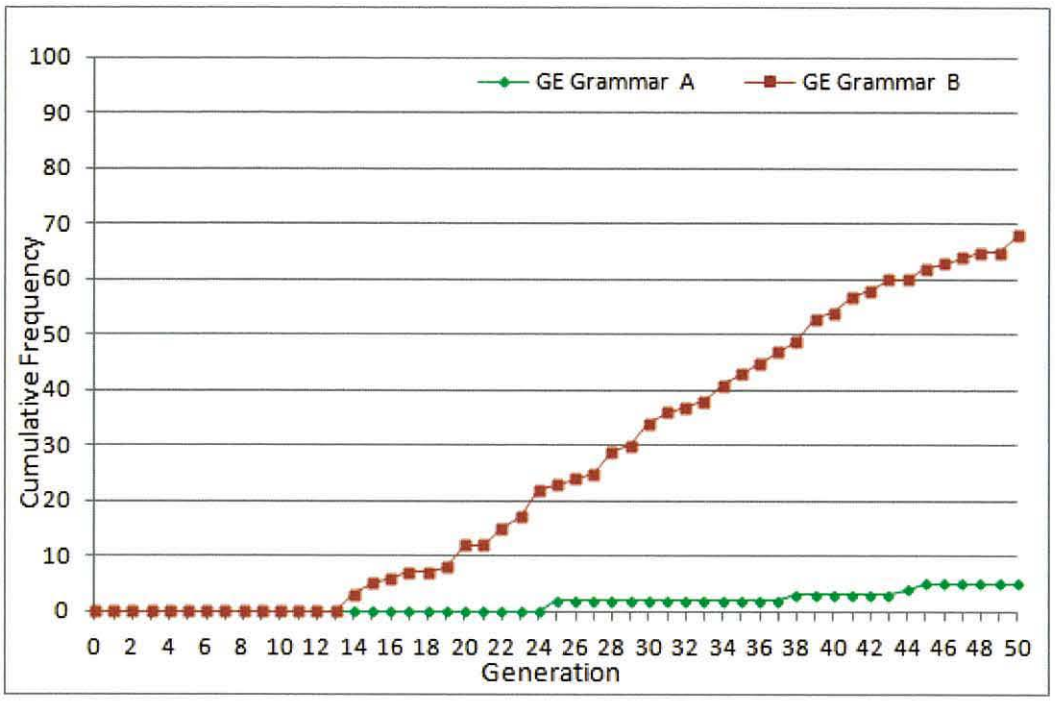


Figure 3.6: Results graph for the symbolic regression problem using GE with BNF grammar (A) and BNF grammar (B).

3.6.5 Trigonometric Identity Experiments

The particular function experimented with was *cos(x)*, and the desired trigonometric identity $1 - 2sin^2x$ (for this reason the Java unary operator *Math.cos()* was not included in the BNF grammar of this problem). The objective of these experiments was to find a mathematical expression identical to the function. The configuration and the BNF grammar definition used in these experiments are shown in Table 3.6 and in Listing 3.9.

Table 3.6: Trigonometric identity GE tableau.

Objective:	Find a new mathematical expression, in symbolic form that equals a given mathematical expression, for all values of its independent variables. Examined Function: *Math.cos(2 * x)*. Desired Trigonometric Identity: $1\text{-}2Sin^2x$.
Terminal Operands:	x, the constant 1.0.
Terminal Operators:	The binary operators +, -, /, and *. The unary operator *Math.sin*.
Fitness cases:	The given sample of the pairs (x_i, y_i) of 20 data points in the interval $[0, 2\pi]$. The input data points (x_i) are randomly created and their corresponding output points (y_i) are automatically created by the expression *Math.cos(2 * x)*.

Raw Fitness:	The sum of the absolute values of errors taken over the fitness cases (x_i, y_i). A lower raw fitness value indicates better individuals.
Standardised Fitness:	Same as raw fitness.
Adjusted Fitness:	$1 / (1 + Fs(i))$ where Fs is the Standardised Fitness of i. The fitness value varies from 0 to 1. Invalid individuals are assigned a zero (0) adjusted fitness value.
Hits:	The number of fitness cases for which the Adjusted Fitness is greater than or equal to 0.999.
Wrapper:	Standard productions to generate a Java class with a main() method which prints the fitness values in the standard output.
Parameters:	Population Size (M) = 500, Maximum Generations (G) = 50, Prob. Mutation (P_m) = 0.01, Prob. Crossover (P_c) = 0.9, Prob. Duplication (P_d) = 0.01, Prob. Pruning (P_p) = 0.01, Steady State GA with Generation Gap (G) = 0.9, Roulette-Wheel Selection, Codon Size = 8.

```
<expr>    ::= <expr> <op> <expr> |
              ( <expr> <op> <expr> ) |
              <pre-op> ( <expr> ) |
              <var>
<op>      ::= + | - | / | *
<pre-op>  ::= Math.sin
<var>     ::= x | 1.0
```

Listing 3.9: BNF grammar used for the trigonometric identity problem.

The results in Table 3.7 show that in a series of 100 evolutionary runs, GE was able to solve the problem.

Table 3.7: Results for the trigonometric identity problem.

	Grammatical Evolution
Evolutionary Runs	100
Highest Fitness Value	1.0
Success Rate	2%

3.7 Experimental Results Discussion

The results of the successful proof-of-concept experiments with jGE confirmed two expected findings. First, that jGE using Grammatical Evolution is able to produce

successful results although not always with a very high success rate. Secondly, that different set-ups and configurations of the searching mechanism and the grammar have a significant impact on the performance (success rate).

The above findings will guide the design and development of future versions of the jGE Library. Namely, the next version of jGE could provide implementations of more genetic operators which will facilitate experiments in a larger range of possible configurations. Also, the need for more than a standard PC's processing power is prominent in order to improve the execution speed of new experiments with jGE. It is well known (Ghanea-Hercock 2003, p.101) that evolutionary algorithms are processing-power demanding algorithms and that they can take advantage of parallel processing architectures. This need for more processing power, in order to reduce the execution speed of the new experiments, could be tackled by incorporating into jGE a parallel distributed processing framework making it possible to execute the problems in question transparently on many machines. Namely, if there are m machines and n individuals, then each one machine will execute genotype-to-phenotype mapping and assign fitness values to n/m individuals.

3.8 Comparison of jGE with other GE Implementations

3.8.1 jGE and libGE

The jGE Library is written in the Java programming language. In contrast, libGE was written in the C++ programming language under GNU/Linux. Therefore jGE can be used in any operating system that provides a Java 5 virtual machine or later which is not the case for libGE. The main architectural and functional difference between jGE and libGE (Nicolau, 2005) is that jGE incorporates the functionality of libGE (grammatical evolution mapping process) as a component and provides its own internal implementation of the libGE's *Search Engine* as well as the *Evaluator*. Namely, jGE is a more general framework for the execution of evolutionary algorithms. Indeed, it still provides, like libGE, the feature of using any other search engine and evaluator beyond that already provided by default in jGE. Also, individual components of the jGE, such as the *GE Mapping Mechanism*, the *BNF Parser*, and the *Mathematical Functions* classes, may also be used separately for special purpose projects.

Another main difference between jGE and libGE is the goal of each project. libGE provides an implementation of the Grammatical Evolution mapping process, whereas the goal of the jGE Library is the development of a general evolutionary algorithms framework which facilitates the incorporation and evaluation of evolutionary techniques; and the incorporation of agent-oriented principles as an extension to develop implementations for parallel distributed systems in forthcoming implementations.

3.8.2 jGE and GEVA

Both jGE and GEVA are implemented with the Java programming language and share a similar architecture which allows the development and addition of modules which specify the search engine to be used, the genetic operators, the evaluation function, the replacement strategy, and other parameters of an evolutionary algorithm. Indeed, GEVA provides a Graphical User Interface (GUI), a Command Line Interface, and many demonstration problems which are not the case for the current version of jGE which provides only an Application Programming Interface (API) and just two out-of-the-box demonstration problems.

The main design and implementation difference between GEVA and jGE is that the former focuses on the implementation of the Grammatical Evolution system. Instead, jGE has been designed as a more general evolutionary algorithms framework of which Grammatical Evolution is just one of the evolutionary algorithms implementations that are currently incorporated. Furthermore, jGE provides a blueprint which supports the future incorporation of aspects like knowledge sharing and environmental factors influences.

Chapter 4

Extensions to jGE

4.1 Introduction

The purpose of this chapter is to describe some extensions to jGE and discuss their experimental results and applicability. Part of the work has been published in Georgiou and Teahan (2006b; 2010).

First, the application of two different approaches of evolving a population of individuals in Grammatical Evolution are investigated, the one using prior knowledge of the domain of the problem in question and the other, using the Darwinian population thinking concept. Both approaches are inspired from nature and are implemented via modification of the BNF grammar definition and the Grammatical Evolution algorithm parameters during the setup of the experiments. These applications demonstrate also the easiness of applying in Grammatical Evolution and jGE, different nature-inspired concepts without necessarily modifying the GE algorithm or the jGE Library code.

Finally, the jGE extension for the NetLogo modelling environment is presented, which enables the use of Grammatical Evolution directly in a NetLogo model. Namely, this extension allows the designer to use the Grammatical Evolution algorithm in NetLogo models for the evolution of the behaviour and/or morphology of agents in a static or dynamic environment.

4.2 Applying prior knowledge and population thinking in Grammatical Evolution

The successful proof-of-concept experiments with jGE (see Chapter 3) have led to further experiments in order to investigate the following:

- The role of prior knowledge in evolutionary runs and its effect on the effectiveness and efficiency of Grammatical Evolution. The expected result in these experiments

is that a restriction of the BNF grammar (implying the use of prior knowledge) should produce better results.

- Demonstration of the usefulness of the evolutionary process for the creation of better solutions. The expected result is that the application of evolutionary mechanisms should produce better results than a random process.
- Initial application of the population thinking principle in evolutionary runs. Promising results on these experiments would lead toward the investigation of the role of the maintenance of the genetic / phenotypic diversity, the creation of different populations (with local optima), and then the combination of individuals from different populations.

Population thinking (Mayr 2002, p.81) was Darwin's radical break with the typological tradition of essentialism of his period. Until then it was believed that seemingly variable phenomena of nature could be sorted into classes. Each class was characterised by its definition (essence). Some people believed that these classes are constant (this school of thought was founded by the Pythagoreans and Plato) and some pre-Darwinian evolutionists (including Lamarck) allowed a gradual change (transformation) of the type (class) over time.

Darwin said (Mayr 2002, p.81) that what we find among living organisms are not constant classes but variable populations. Every species is composed of numerous local populations and the unit of evolution is the population (not species). Darwin's new way of thinking, being based on the study of populations, is now referred to as "population thinking".

Population thinking favours the acceptance of gradualism and is one of the most important concepts in biology. It is the foundation of modern evolutionary theory and one of the basic constituents of the philosophy of biology (Mayr 2002, p.81). Consequently, the incorporation of population thinking in Grammatical Evolution (or any other evolutionary algorithm) could possibly lead towards extremely interesting results and conclusions and therefore it is believed by Georgiou and Teahan (2006b) that such an investigation is worthy. The results of some first initial experiments toward this approach are very promising. These are described and discussed (alongside with other ideas) in the following sections.

4.3 Prior Knowledge Experiments

In the experimental setups of this section, the BNF grammars of the previous chapter for the symbolic regression (Listing 3.7 and Listing 3.8) and trigonometric identity (Listing 3.9) problems have been modified, thereby restricting in this way the possible phenotypes that are produced and guiding the evolutionary algorithm. The development and creation of a more suitable BNF grammar implies the use of prior knowledge (at least in a rudimentary sense) with the aim to restrict the search space of the problem in question. This is directly related to the "I" part of what Koza (2003) names *AI ratio* (the "artificial-to-intelligence" ratio) of a problem-solving method. He defines the *AI ratio* in Koza (2003) as "the ratio of that which is delivered by the automated operation of the artificial method to the amount of intelligence that is supplied by the human applying the method to a particular problem".

For the symbolic regression using prior knowledge experiment, 100 evolutionary runs have been performed. Table 4.1 shows the setup and configuration of the experiment and Listing 4.1 shows the BNF grammar which has been modified in order to restrict the search space. Comparing to the grammar of Listing 3.7 (grammar A), the <*pre-op*> and its productions have been removed and further reductions have been made in the productions of the <*expr*> and <*op*> non-terminal symbols. Against the grammar of Listing 3.8 (grammar B), the only change is the removal of the subtraction (-) and division (/) terminal symbols from the non-terminal symbol <*op*>.

Table 4.1: Symbolic regression GE tableau.

Objective:	Find a function of one independent variable and one dependent variable, in symbolic form that fits a given sample of 20 (x_i, y_i) data points, where the target function is the quadratic polynomial $x^4 + x^3 + x^2 + x$.
Terminal Operands:	x (the independent variable), the constant *1.0*.
Terminal Operators:	The binary operators + and *.
Fitness cases:	The given sample of the pairs (x_i, y_i) of 20 data points in the interval [-1, +1]. The input data points (x_i) are randomly created and their corresponding output points (y_i) are automatically created by the expression $x^4 + x^3 + x^2 + x$.
Raw Fitness:	The sum of the absolute values of errors taken over the fitness

	cases (x_i, y_i). A lower raw fitness value indicates better individuals.
Standardised Fitness:	Same as raw fitness.
Adjusted Fitness:	$1 / (1 + Fs(i))$ where Fs is the Standardised Fitness of i. The fitness value varies from 0 to 1. Invalid individuals are assigned a zero (0) adjusted fitness value.
Hits:	The number of fitness cases for which the Adjusted Fitness is greater than or equal to 0.999.
Wrapper:	Standard productions to generate a Java class with a main() method which prints the fitness values in the standard output.
Parameters:	Population Size (M) = 500, Maximum Generations (G) = 50, Prob. Mutation (P_m) = 0.01, Prob. Crossover (P_c) = 0.9, Prob. Duplication (P_d) = 0.01, Prob. Pruning (P_p) = 0.01, Steady State GA with Generation Gap (G) = 0.9, Roulette-Wheel Selection, Codon Size = 8.

```
<expr> ::= <expr> <op> <expr> |
           <var>
<op>   ::= + | *
<var>  ::= x
```

Listing 4.1: BNF grammar definition for the symbolic regression experiment using prior knowledge.

Table 4.2 shows the results of the experiment and Figure 4.1 depicts the cumulative frequency measure of success over 100 runs of GE using prior knowledge against the symbolic regression setups of Chapter 3 which use the grammars of Listing 3.7 (grammar A) and Listing 3.8 (grammar B). The results show a dramatic improvement of the success rate. Namely, using prior knowledge GE achieves the absolute success rate (100%) against 5% (grammar A). Note also that the grammar of Listing 3.8 (grammar B), with which GE achieves success rate 68%, implies also the use of prior knowledge but less than in the case of the grammar which is used in the setup of this section (Listing 4.1).

Table 4.2: Results for symbolic regression using prior knowledge.

	GE using Prior Knowledge
Evolutionary Runs	100
Highest Fitness Value	1.0
Success Rate	100%

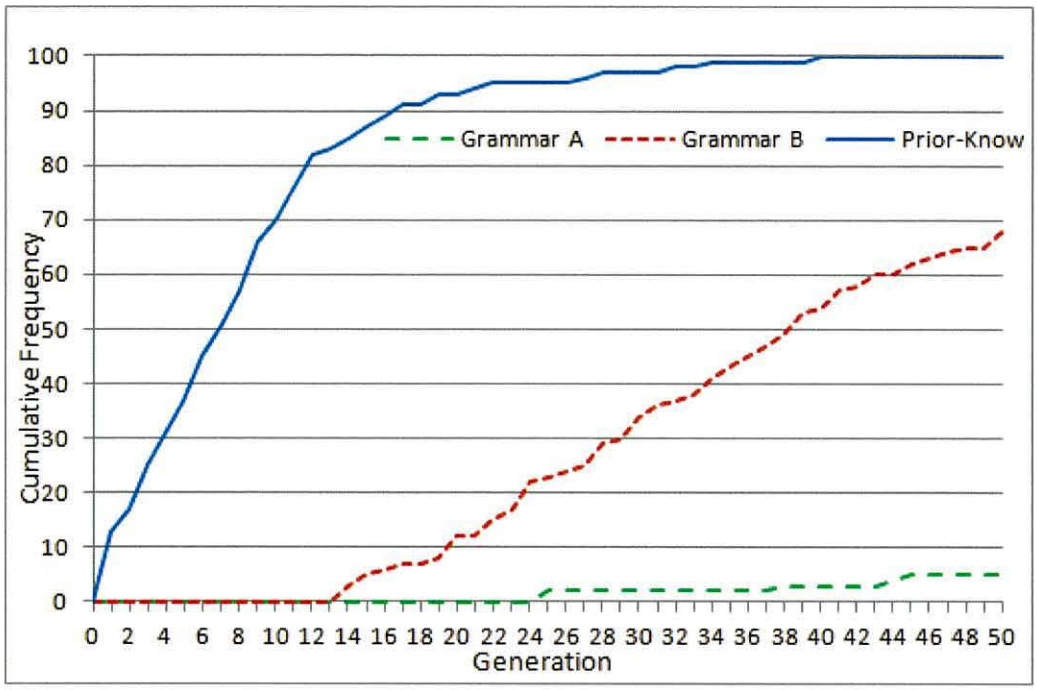


Figure 4.1: Comparison graph for symbolic regression (using prior knowledge).

The configuration of the trigonometric identity experiment can been seen in Table 4.3. In this experiment, 100 evolutionary runs were performed using the grammar of Listing 4.2. The only change in the BNF grammar compared to the grammar of Listing 3.9 is the replacement of the production *<pre-op> (<expr>)* with *<pre-op> (<var>)* thereby not allowing the recursive call of the Java method *Math.sin()* or any other expression to be passed as arguments in *Math.sin()* except the constant *1.0* or the variable *x*.

Table 4.3: Trigonometric identity GE tableau.

Objective:	Find a new mathematical expression, in symbolic form that equals a given mathematical expression, for all values of its independent variables. Examined Function: *Math.cos(2 * x)* Desired Trigonometric Identity: $1\text{-}2Sin^2x$.
Terminal Operands:	*x*, the constant *1.0*.
Terminal Operators:	The binary operators +, -, /, and *. The unary operator *Math.sin*.
Fitness cases:	The given sample of the pairs (x_i, y_i) of 20 data points in the interval $[0, 2\pi]$. The input data points (x_i) are randomly created and their corresponding output points (y_i) are automatically created by the expression *Math.cos(2 * x)*.
Raw Fitness:	The sum, of the absolute values of errors taken over the fitness cases (x_i, y_i). A lower raw fitness value indicates better

	individuals.
Standardised Fitness:	Same as raw fitness.
Adjusted Fitness:	$1 / (1 + Fs(i))$ where Fs is the Standardised Fitness of i. The fitness value varies from 0 to 1. Invalid individuals are assigned a zero (0) adjusted fitness value.
Hits:	The number of fitness cases for which the Adjusted Fitness is greater than or equal to 0.999.
Wrapper:	Standard productions to generate a Java class with a main() method which prints the fitness values in the standard output.
Parameters:	Population Size (M) = 500, Maximum Generations (G) = 50, Prob. Mutation (P_m) = 0.01, Prob. Crossover (P_c) = 0.9, Prob. Duplication (P_d) = 0.01, Prob. Pruning (P_p) = 0.01, Steady State GA with Generation Gap (G) = 0.9, Roulette-Wheel Selection, Codon Size = 8.

```
<expr>   ::= <expr> <op> <expr> |
             ( <expr> <op> <expr> ) |
             <pre-op> ( <var> ) |
             <var>
<op>     ::= + | - | / | *
<pre-op> ::= Math.sin
<var>    ::= x | 1.0
```

Listing 4.2: BNF grammar definition for the experiments of the trigonometric identity problem with prior knowledge.

A dramatic improvement of the success rate has been achieved in the trigonometric identity problem as well. Namely, the success rate rose from the 2% of the setup of the previous chapter (see Table 3.7) to 47% (see Table 4.4). Figure 4.2 depicts the cumulative frequency measure of success over 100 runs of Grammatical Evolution using prior knowledge against the trigonometric identity setup of Chapter 3.

Table 4.4: Results for the trigonometric identity problem using prior knowledge.

	Trigonometric Identity with Prior Knowledge
Evolutionary Runs	100
Highest Fitness Value	1.0
Success Rate	47%

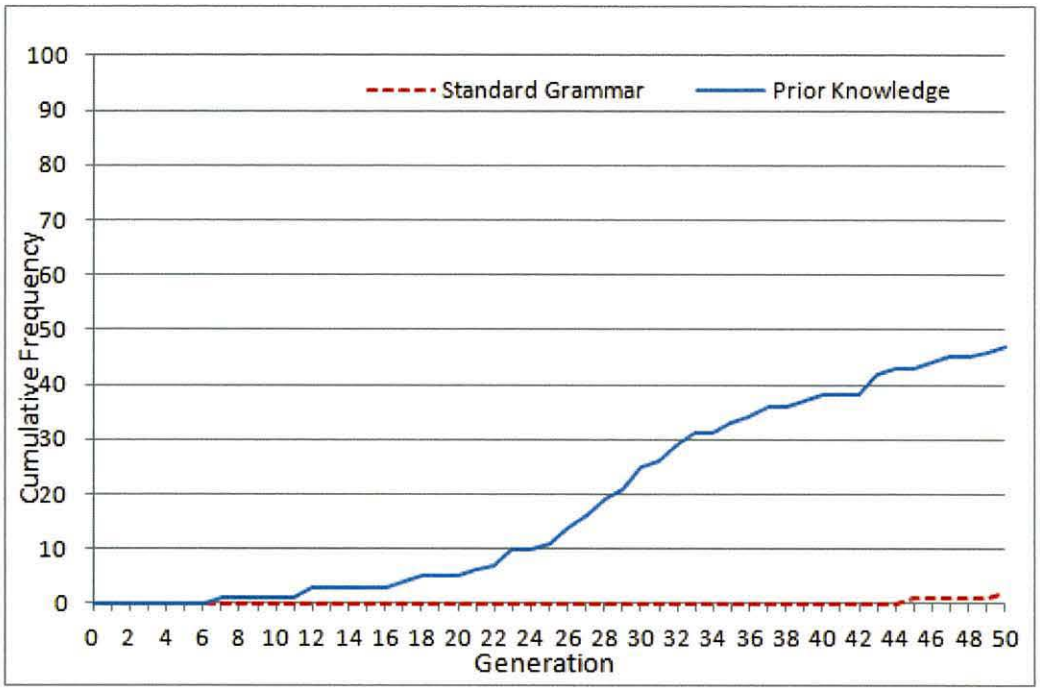


Figure 4.2: Comparison graph for the trigonometric identity problem (using prior knowledge).

The results of the above experimental setups demonstrate that the incorporation of prior knowledge (namely increasing the "I" of the *AI ratio*) and consequently restricting the search space, can be achieved with success in jGE very easily and quickly through just the modification of the BNF grammar. In this way, domain knowledge of the problem in question can be incorporated and utilised very easily – in order to improve the performance of the evolutionary algorithm – which is one of the strongest characteristics and advantages of the Grammatical Evolution algorithm.

4.4 Population Thinking Experiments

According to Darwin (1859), what we find among living organisms in nature are not constant classes but *variable populations*. Indeed, every *species* is composed of numerous local populations and the unit of evolution is the population, not the species (Darwin, 1859; Mayr 2002, p.81). Consequently, closer in a sense to the Darwin's population thinking concept than the standard Genetic Algorithm seems to be the *island* parallel GA which considers relatively isolated *demes* (Cantú-Paz, 1998). A *deme* in the *island* parallel GA model could be thought that represents a *variable population* and the population of the standard GA that represents a *species*.

The setups of the trigonometric identity experiments of this section simulate a random search (setup 1) and – in a rudimentary sense – the *island* parallel GA model (setup 2 and setup 3). The configuration and the BNF grammar of these setups are the same with those of the trigonometric identity problem of the previous section (see Table 4.3 and Listing 4.2) except the *Maximum Generations (G)* and *Population Size (M)* parameters which are listed below for each setup. The BNF grammar is depicted also in Listing 4.3 for clarity. Note that in all setups (including the "Standard" setup of the previous section), the product of *total number of evolutionary runs*, *maximum generations*, and *population size* is the same in order to make the results comparable:

- Setup 1 (Random Search): 5000 runs with *Max Generations (G)* set at 1 and *Population Size (M)* at 500. The best individual is selected from each run.
- Setup 2 (Many Populations with Few Generations): 100 runs of 10 sub-runs with *Max Generations (G)* set at 5 and *Population Size (M)* at 500. For each of the 100 runs, the best individual is selected from the 10 best individuals from each sub-run.
- Setup 3 (Many Small Populations - Population thinking): 100 runs of 10 sub-runs with *Max Generations (G)* set at 50 and *Population Size (M)* at 50. For each of the 100 runs, the best individual is selected from the 10 best individuals from each sub-run.

```
<expr>   ::= <expr> <op> <expr> |
             ( <expr> <op> <expr> ) |
             <pre-op> ( <var> ) |
             <var>
<op>     ::= + | - | / | *
<pre-op> ::= Math.sin
<var>    ::= x | 1.0
```

Listing 4.3: BNF grammar definition for the evolutionary process evaluation and population thinking experiments (trigonometric identity problem). Note that it is the same grammar with that of Listing 4.2.

Table 4.5 shows the results of the experiments using the mentioned setups (setup 1, setup 2, and setup 3) and Figure 4.3 depicts the cumulative frequency measure of success over 100 runs of the "Few Generations" (setup 2) and "Many Small Populations" (setup 3) Grammatical Evolution setups against the "Standard" Grammatical Evolution setup of the previous section (see configuration of Table 4.3 and results in Table 4.4).

Table 4.5: Evolutionary process evaluation and population thinking experimental results (trigonometric identity problem).

	Random Search	Few Generations	Many Small Populations
Runs	5000	100	100
Highest Fitness	1.0	1.0	1.0
Success Rate	0.18%	7%	30%

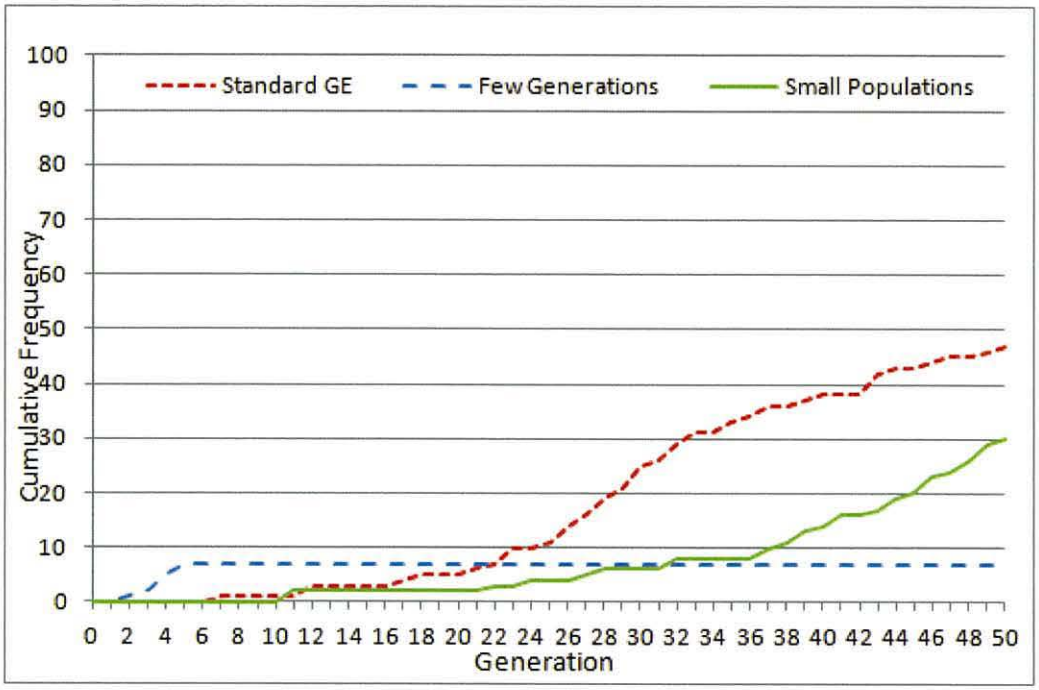


Figure 4.3: Population thinking experiments results graph.

The low success rates of random search (0.18%) and "Few Generations" setup (7%) compared to the success rate of the "Standard" Grammatical Evolution setup of the previous section (47%) demonstrate the usefulness of the evolutionary process and its superiority over a random process which is based on the probability of the creation of a "more suitable" initial population. This is also supported by the superiority of the "Many Small Populations" (population thinking) setup (30%) over the "Few Generations" (7%). It is shown that a long evolution period (50 generations) of small populations (50 individuals) produces better results than a short evolution period (5 generations) of large populations (500 individuals). But the most promising result of these experiments is that the population thinking approach ("Many Small Populations") displayed a success rate (30%) closer to the "Standard" setup of one population (47%) than the other two setups even though the former uses a much smaller evolvable genetic pool because of the

complete isolation of small populations (demes). This encourages the further investigation of the population thinking approach and its combination with other evolutionary principles.

4.5 The jGE NetLogo Extension

NetLogo (Wilensky, 1999) is a multi-agent programmable modelling environment, written in the Java programming language, for simulating natural and social phenomena. Modellers can give instructions to as many "agents" (turtles, patches, and links) they want, using a variation of the Logo modelling language. The agents are operating independently in a simulation and the modeller can observe the collective results of all agents' behaviour.

One of the many features of NetLogo is the provisioning of an extensions API which allows the addition of new commands and reporters to the NetLogo language. The extensions can be developed in Java and other programming languages such as Scala. The jGE NetLogo extension is an implementation of the Grammatical Evolution mapping process for the NetLogo modelling environment (NetLogo 4.1.x), written in Java, and has been developed as part of the work described in this text.

The jGE NetLogo extension lets users of NetLogo incorporate within their models a small part of the functionality and features of the jGE Library (Georgiou and Teahan, 2006a; 2006b; 2008). Namely, it provides primitives which allow the users to take advantage of the Grammatical Evolution algorithm and utilise it for the evolution of the morphology and/or the behaviour of NetLogo agents (turtles). This extension includes reporters and commands which provide the functionality of Grammatical Evolution plus some supporting / helper primitives mentioned below.

The goal of this extension is to allow both NetLogo users to get familiar with and use Grammatical Evolution within their models, and users interested in Evolutionary Computation to use evolutionary algorithms (like Grammatical Evolution) directly within a simulation environment for the evolution of the morphology and behaviour of agents.

Furthermore, another goal of this extension is to facilitate the evolution of the behaviour of NetLogo agents. The key factor in most of the Evolution models of NetLogo is how the

agent-environment interaction affects the simulation. There are two components to this (fairly obviously) – the agents, and the environment. What the existing NetLogo agents do is interact with (possibly) a dynamic environment. Usually the agents themselves are not dynamic themselves – i.e. their behaviour does not usually alter based on the changing environment. What the jGE NetLogo extension provides is the ability for the human modeller to alter the behaviour of the agents. The agents themselves can also alter the environment, and with the jGE NetLogo extension, it is now possible to have the environment affect the resultant behaviour of the agents as they determine what is best to interact successfully (or even survive) within it.

The main procedures (commands and reporters) provided by the jGE NetLogo extension are listed in Appendix G. The complete documentation can be downloaded from Georgiou (2006) or can be found in the accompanying CD.

The main usage pattern of jGE in NetLogo models is as follows: The user creates a model for the problem in question and defines the agents (breeds), and their initial morphology and/or behaviour. The domains of these are defined in external text files as BNF grammar/s which will be used for the mapping of the genotype of the agents (binary string) to the phenotype (NetLogo agent's attributes or actions). In the case of the evolution of the morphology of an agent, the modeller has to assign the evolved attributes to the agent by decoding the resultant phenotype and using the NetLogo programming language features. In the case of the evolution of the behaviour of an agent, the modeller has to execute the resultant behaviour (phenotype) using the "*run*" NetLogo native primitive.

Also, regarding the selection mechanism, and the replacement strategy, these are responsibilities of the NetLogo model, so consequently they have to be implemented in the model using the NetLogo programming language. In this way the modellers gain the maximum flexibility (limited only by the NetLogo features) plus the advantage of using the Grammatical Evolution algorithm in a straight and simple manner for the evolution of their agents.

Listing 4.4 shows a NetLogo code sample of using the jGE NetLogo extension.

```
(1)  jge:load-bnf "BNFDemo.bnf" "bnf_demo"
(2)
(3)  let genotype jge:individual 4 5 10
(4)  ; A random genotype:00100000111011111000011011110111010 1
(5)
(6)  let action jge:phenotype genotype "bnf_demo" 4 10
(7)  ; The corresponding phenotype: rt 90 fd 1 fd 1 lt 90
(8)
(9)  ; create a turtle and give it commands
(10) crt 1
(11) ask turtle 0 [set heading 0]
(12) ask turtle 0 [run action]
```

Listing 4.4: Sample code of using the jGE NetLogo extension.

In line (1), the BNF grammar definition *bnf_demo* shown in Listing 4.5 is loaded. This grammar definition dictates legal NetLogo code which defines the actions of a turtle (an agent that move around in the world). Line (3) sets to the variable *genotype* a randomly created binary string of codon size *4*. The size of the genotype will be between *5* and *10* codons. In line (6), the *genotype* is mapped to the corresponding phenotype using the *bnf_demo* grammar, and specifying that the codon size is *4* and the GE wrapping limit is *10*. The resulted phenotype is stored in the variable *action*. Lines (10) and (11) create a turtle facing north. Line (12) commands the turtle to execute the phenotype (*actions*).

```
<action>     ::= <action> <move> |
                 <move>
<move>       ::= <forward> |
                 <turn-left> |
                 <turn-right>
<forward>    ::= fd 1
<turn-left>  ::= lt 90
<turn-right> ::= rt 90
```

Listing 4.5: Sample BNF grammar definition for a NetLogo turtle.

Finally, it is worth noting that the jGE NetLogo extension is the only Grammatical Evolution extension for NetLogo and the only application of Grammatical Evolution in the NetLogo modelling environment. It is freely available from both the jGE web site (see Georgiou, 2006) and the NetLogo web site (see Wilensky, 1999).

4.6 Experimental Conclusions and Discussion

The promising results from the initial investigation of using prior knowledge and population thinking reveal an additional potential area of investigation. Taking into

account that evolution on Earth is guided by chance and determination (Mayr 2002, p.132), it would be interesting to find out if the application of Evolutionary Synthesis (Ridley 2004, pp.14-18) principles (such as population thinking, microevolution, macroevolution, elimination pressure, common ancestor, species, and more) could result in a self-emerged evolutionary system instead of an explicitly pre-determined one (as is the case in Genetic Algorithms where the reproduction and diversity mechanisms of nature such as crossover and mutation are given in advance). Namely, is it possible for a computer, based on chance (pseudo-randomness) and determination (application of evolutionary and genetics principles in machines), to emerge an evolutionary system which is able to evolve useful programs, solutions, and agents?

Another example of the application of evolution principles, apart from population thinking, is the species concept (Mayr 2002, pp.180-182). It is widely accepted today that the isolation mechanisms of species are devices to protect the integrity of well-balanced, harmonious genotypes, and the integrity of species is maintained by natural selection (Mayr 2002, p.186). In light of this theory, the use of mechanisms where species of individuals emerge for the maintenance of good solutions and their gradual improvement without the danger of destroying them, could be tested.

One more idea of further investigation is the utilisation of knowledge as a fitness value factor. Because, according to Neo-Darwinism, there are no heritable acquired characteristics (Mayr 2002, p.95-96), it could be assumed that the Shared Knowledge is a factor which affects the Natural Selection process and allows existing or new genetic/phenotypic characteristics to posses different importance and value (fitness). Namely, that the evolving Shared Knowledge will favour different phenotypes (as happens in nature with the changes in the environment).

Furthermore, the population thinking principle can be extended in a Family Based approach (Teahan, Al-dmour and Tuff, 2005) by incorporating Knowledge Sharing and simulating in this way social phenomena of living organisms and especially humans (e.g. families). These experiments will show in which degree phenotypic variation and evolution/sharing of knowledge could lead to better and faster solutions in the area of evolutionary algorithms.

Finally, the incorporation of a subset of the jGE functionality in the NetLogo modelling environment – with the implementation of the jGE NetLogo Extension – facilitates the evolution of the behaviour of NetLogo agents using the Grammatical Evolution algorithm. Also, jGE in contrast to libGE and GEVA, provides a more general framework for the implementation and extension of evolutionary algorithms which means that it can be used as a tool for further experimentation in evolutionary algorithms, besides Grammatical Evolution, using additionally a programmable modelling environment such as NetLogo. The next chapter introduces the application of Grammatical Evolution in the Santa Fe Trail problem using the jGE library and the NetLogo modelling environment, and then presents and discusses the experimental results.

Chapter 5

Grammatical Evolution and the Santa Fe Trail Problem

5.1 Introduction

In this chapter, the results of a series of experiments that explore the effectiveness of Grammatical Evolution for the Santa Fe Trail problem are presented. The experiments and their results have been published in Georgiou and Teahan (2010) and support the claim of Robilliard, et al. (2006) that the comparison mentioned in the Grammatical Evolution literature (O'Neill and Ryan 2001; 2003, pp.55-58) between Grammatical Evolution and Genetic Programming regarding the Santa Fe Trail problem is not a fair one.

Namely, even though GE literature claims that Grammatical Evolution outperforms Genetic Programming in the Santa Fe Trail problem, it is experimentally proved in the following sections that this happens only because the experiments described in the GE literature use a different and narrower search space. Also, it is shown in this chapter that Grammatical Evolution is capable of finding solutions in the Santa Fe Trail problem that require fewer steps than the solutions mentioned in the Genetic Programming or Grammatical Evolution literature (Koza, 1992; O'Neill and Ryan, 2003) if a wider search space is used than that defined in the GE literature.

For the execution of the experiments, a series of tools and models have been used: a) jGE, a Java implementation of the Grammatical Evolution system (see Chapter 3); b) jGE NetLogo, an extension of jGE for the NetLogo modelling environment (see Chapter 4); c) the Santa Fe Trail model, a simulation of the problem in NetLogo; and d) a NetLogo model for the simulation of Grammatical Evolution evolutionary runs in the Santa Fe Trail problem. Both NetLogo models have been developed to facilitate the execution of these experiments and are described in detail in this chapter.

5.2 The Grammatical Evolution Issue

Robilliard, et al. (2006) say that in order to evaluate Genetic Programming algorithms using the Santa Fe Trail problem as a benchmark "the search space of programs must be semantically equivalent to the set of programs possible within the original Santa Fe Trail definition." However, it has been argued by Robilliard, et al. (2006) that in GE literature, Genetic Programming (Koza, 1992) and Grammatical Evolution (O'Neill and Ryan, 2003) have not been compared in the Santa Fe Trail problem using semantically equivalent search spaces. Namely, that Grammatical Evolution is benchmarked against GP using a grammar that states a declarative language bias which restricts the original search space and moreover (as noted below) excludes good solutions in terms of required steps.

Furthermore, Robilliard, et al. (2006) note about the upper limit of the time steps allowed for an ant to execute, that Koza arbitrarily fixed to 400 but Langdon and Poli assumed a possible mistake in the original Koza's work, so the later set the maximum time limit to 600 steps. Also, Robilliard, et al. (2006) argue that they found no solution in the Santa Fe Trail problem using Grammatical Evolution, up to and including 600 time steps in their experiments, and they note that almost all Grammatical Evolution publications mention a maximum of 615 time steps allowed which is different from both Koza and Langdon's settings. Only in O'Neill and Ryan (2001) with Grammatical Evolution is the limit reported as 600 steps, which Robilliard, et al. (2006) claim is a mistype.

The above claims about the mistypes regarding the maximum allowed steps mentioned in Koza (1992) and O'Neill and Ryan (2001) are also supported by the experimental results of this chapter where it is observed through simulations that the best Genetic Programming solution mentioned in Koza (1992, p.154) needs 545 steps, and the best Grammatical Evolution solution mentioned in O'Neill and Ryan (2001; 2003, p.56) needs 615 steps to complete, which exceed the 400 and 600 maximum steps limits respectively.

5.3 NetLogo Models

5.3.1 Santa Fe Trail Model

This model (see Figure 5.1) is a simulation of the Santa Fe Trail problem. It has been developed – as a subproject of the jGE project – in order to enable the modeller to do the following things:

- Write a set of commands in the NetLogo programming language which control the actions of an artificial ant, and run them to observe its behaviour.
- Simulate, verify, and investigate the Santa Fe Ant Trail solutions given by relevant publications or found by other related programs (for example, some Genetic Programming and Grammatical Evolution software packages).

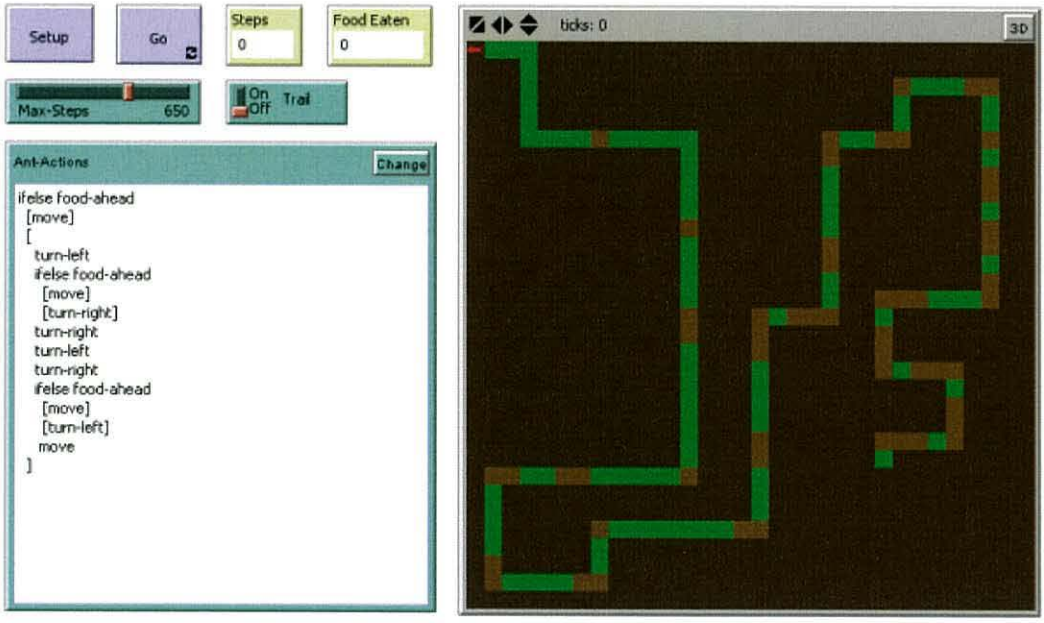


Figure 5.1: Interface of the Santa Fe Trail NetLogo model.

This NetLogo simulation is – to the author's knowledge – the only Santa Fe Trail implementation publicly available, that allows the user to run and investigate a given solution through a GUI. It is freely available from both the jGE (Georgiou, 2006) and the NetLogo (Wilensky, 1999) web sites. A variation of the same model is available at files.bookboon.com/ai.

The following guidelines of how to use the model demonstrate the easiness of executing a simulation of an ant's control program to the Santa Fe Trail problem.

How to use the model

First, the model's speed must be set to normal (by adjusting the speed slider in the toolbar). This will let the setup procedure redraw the path very quickly. Then, the "Setup" button is pressed to initialise the model's world. After that, the Santa Fe Trail will appear in the 2D view of the NetLogo interface. Indeed, the multi-line input box "Ant-Actions" is initialised with one of the Koza's solutions (1992, p.154) in the Santa Fe Trail problem, written in the NetLogo programming language. The code of the "Ant-Actions" box is editable in order to be able the modeller to write and enter for execution in the model custom ant control programs. Listing 5.1 shows the valid actions of an ant (NetLogo commands and reporter), and a sample ant control program which dictates the ant to move forward one square if there is food ahead otherwise to turn right.

```
NetLogo ant commands: move turn-right turn-left
NetLogo ant reporter    : food-ahead

Sample Control Program:
ifelse food-ahead
  [move]
  [turn-right]
```

Listing 5.1: Artificial ant actions and a sample control program, in the NetLogo programming language.

The commands of the "Ant-Actions" box will be executed by the ant when the "Go" button is pressed. Before the simulation starts, the "Max-Steps" slider must be set to the maximum number of steps (time units) that are allowed to be executed by the ant during the simulation. These steps should not be confused with the NetLogo's discrete steps called "ticks". Also, by turning-on the switch "Trail" the path followed by the artificial ant during the simulation will be coloured. It is advised that the model's speed be turned to very slow (with adjusting the speed slider in the toolbar) in order for the ant's movements to be observable by the modeller. Otherwise the ant will run so fast that the modeller will actually see only the end results of the simulation. After the required configuration, the simulation can start with pressing the "Go" button. During the

simulation, the monitors "Steps" and "Food Eaten" display the current number of the executed actions and the pieces of food eaten by the ant so far, respectively.

Model validation

In order that the NetLogo implementation of the Santa Fe Trail problem be tested and validated, the results of some of the programs mentioned by Koza (1992) were compared with the corresponding results of the NetLogo model.

The following programs found in Koza (1992, p.151) were executed in the model and compared with the raw fitness reported by Koza: The "move" program; the "quilter" program; and the "avoider" program. The LISP code of these programs, the corresponding NetLogo code, and the Koza's & NetLogo results are depicted in Table 5.1. The last two columns of the table indicate that the NetLogo simulation gave the same results with these reported by Koza.

Table 5.1: Comparing LISP and NetLogo solutions of the Santa Fe Trail.

Program	LISP code	NetLogo Code	Koza Raw Fitness	NetLogo Raw Fitness
Move	`(PROGN2 (MOVE) (MOVE))`	`move` `move`	3	3

Quilter	`(PROGN3 (RIGHT)`
	`(PROGN3 (MOVE) (MOVE)`
	`(MOVE))`
	`(PROGN2 (LEFT) (MOVE)`
	`) )`
		`turn-right`
		`move`
		`move`
		`move`
		`turn-left`
		`move`
			Finds 4 pieces of food in the first vertical cross of the grid.	Finds 4 pieces of food in the first vertical cross of the grid.

Avoider	`(IF-FOOD-AHEAD (RIGHT)`
	`(IF-FOOD-AHEAD`
	`(RIGHT)`
	`(PROGN2 (MOVE)`
	`(LEFT) ) ) )`
		`ifelse food-ahead`
		`[turn-right]`
		`[`
		`ifelse food-ahead`
		`[turn-right]`
		`[move turn-left]`
		`]`
			0	0

Additionally, the NetLogo implementation was tested with more solutions which are either provided by the GE literature or were generated by GEVA (UCD, 2008). The comparison of the results of the execution of these solutions in the NetLogo simulation with those already reported, verified again the correctness of the NetLogo implementation of the Santa Fe Trail simulation.

5.3.2 Evolutionary Runs in SFT Model

Model overview

This NetLogo model simulates a series of evolutionary runs for the Santa Fe Trail problem using the Grammatical Evolution genotype-to-phenotype mapping process, a genetic algorithm as the search engine, roulette wheel as the selection mechanism of the parents, and a steady state replacement strategy for the creation of the offspring population. The model uses the jGE NetLogo extension for the Grammatical Evolution mapping; all other parts have been implemented with the NetLogo programming language in the model.

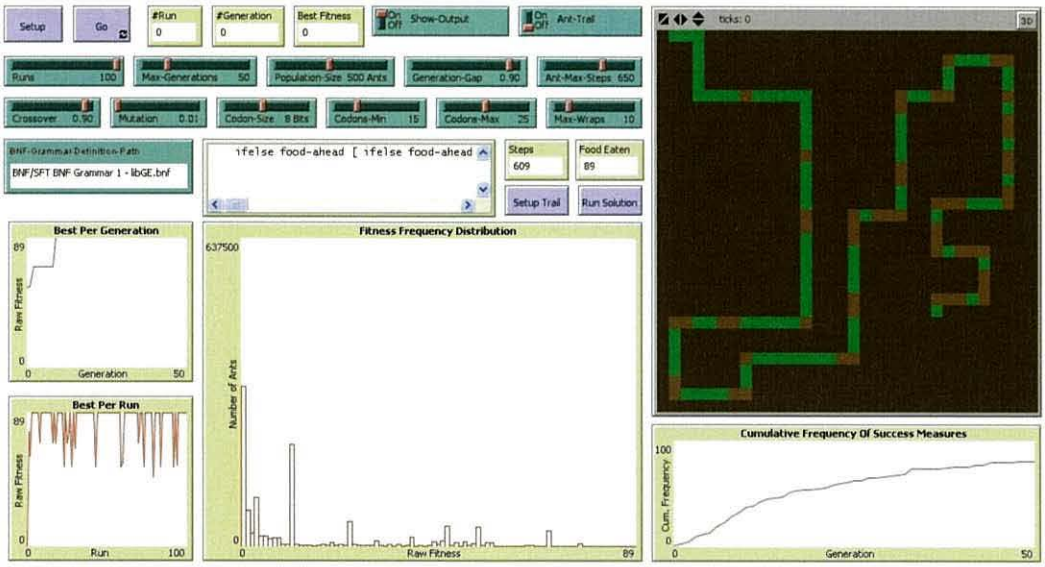


Figure 5.2: Interface of the Evolutionary SFT model for GE.

A screenshot of this model is shown in Figure 5.2. It depicts the Santa Fe Trail drawn in NetLogo's environment on the right top of the Interface, there are buttons and sliders for configuring the Grammatical Evolution algorithm and setting up the experiment, executing and defining the simulation on the top left, and graphs for tracking the state of the simulation are shown in the middle left and at the bottom.

This model is the only published and widely available NetLogo model that uses an evolutionary algorithm (Grammatical Evolution in this case) to evolve solutions of the Santa Fe Trail problem. It provides a useful tool for studying aspects of evolutionary algorithms, Grammatical Evolution, artificial ant problem and NetLogo modelling.

How the model works

The button "Setup" initialises the environment. When the button "Go" is pressed, the specified number of runs (evolutionary runs) is executed. Each run is independent from the others. Namely, a new random population of ants is created when a run starts which consist the generation zero of this run. The range of the sizes of the genomes of the ants of the initial population is determined by the size of the codon in bits and by the allowed range of codons for the genotype of an individual. Note here that this limit does not apply to the genomes of the offspring. During a run, the population of ants is evolved for the specified number of generations and the size of the population (number of ants) remains constant.

For the evolution of the population of the ants, the search engine used by Grammatical Evolution in this model is a steady state Genetic Algorithm. The main idea of a steady state Genetic Algorithm is that a portion of the population P survives in the new population P' and that only the worst individuals are replaced. Namely, a few good individuals, selected using the roulette wheel selection mechanism, will mate through crossover and their offspring which are subject to mutation, will replace the worst individuals. The rest of the population will survive. The "Generation Gap" determines the portion of the population that will be replaced. A detailed description of the steady state Genetic Algorithm can be found in Chapter 3 as well as in Georgiou (2006) and in Georgiou and Teahan (2008).

The evaluation of the evolved ants is performed sequentially. Namely, for each ant of the population, the Grammatical Evolution genotype-to-phenotype mapping process is applied to the genotype using the given BNF grammar definition and wrapping the genome if required, up to the specified maximum allowed number of wraps. Then, for each valid individual (an individual with phenotype that does not contain non-terminals), a Santa Fe Trail simulation takes place in order to calculate for this ant the fitness value. The visual depiction of this simulation appears to the right part of the screen (2D View). Consequently, the evaluation of a generation will require as many independent Santa Fe Trail simulations as the size of the population. The raw fitness value of an ant is the number of food found and eaten by this ant during the simulation until the maximum number of allowed moves (ant steps) is reached.

During the evolutionary runs, useful statistics appear in the plots of the Interface Tab of the model. The "Best per Generation" plot shows the fitness value of the best ant per generation of the current run. The "Best per Run" plot shows the fitness value of the best ant per run. The "Fitness Frequency Distribution" plot shows the distribution of the fitness values of all ants created so far. The "Cumulative Frequency of Success Measures" plot shows the cumulative frequency of success measures of all runs of the current experiment. A cumulative frequency plot is a way to display cumulative information graphically, namely it shows the number, percentage, or proportion of observations in a data set that are less than or equal to particular values. For example, the cumulative frequency for a value x is the total number of scores that are less than or equal to x. In this particular model, it shows the number of evolutionary runs in which a solution has been found before or at a number of population generations.

The monitor fields of the model interface show the number of the current run, the number of the current generation of this particular run, and the fitness of the best ant found so far in the current run.

When the model finishes the execution (namely the specified number of evolutionary runs is performed), the best solution of all runs appears in the "Output" control in the middle of the left part of the screen. In addition, the fitness value (pieces of food eaten) and the required moves (steps) appear in the corresponding monitor controls. The control program of this ant (solution) can be executed again for study by pressing in sequence the "Setup Trail" and the "Run Solution" buttons. The program can be executed as many times as needed by the modeller, as long as these buttons are pressed in the same order as mentioned and the button "Setup" is not pressed, because the later will initialise the experiment. In order to watch and study the behaviour of the ant during a simulation, the speed slider must be moved to the left so that the simulation is slowed down.

5.4 Experiments Setup

A series of experiments has been conducted using two different BNF grammar definitions. The first, named here BNF-Koza (Listing 5.2), is a translation to the NetLogo programming language of the SFT-BAP grammar cited in Robilliard, et al. (2006). This grammar defines a search space semantically equivalent to Koza's original

implementation search space (Robilliard, et al., 2006). The second grammar definition, named here BNF-O'Neill (Listing 5.3), is the translation to the NetLogo programming language of the BNF grammar definition mentioned in the Grammatical Evolution literature (O'Neill and Ryan, 2001; 2003, p.55) regarding the Santa Fe Trail problem. When the Grammatical Evolution algorithm uses the BNF-O'Neill grammar, it will be referred as "GE using BNF-O'Neill". In the same way, when it uses the BNF-Koza grammar, it will be referred as "GE using BNF-Koza".

The tuples $\{N, T, P, S\}$ of the BNF-Koza and the BNF-O'Neill grammar definitions are depicted in Listing 5.2, and in Listing 5.3 respectively, where N is the set of non-terminal symbols, T the set of terminal symbols, P a set of production rules that maps the elements of N to T, and S is a start symbol that is a member of N.

```
N = {expr, line, op}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = expr

P:
<expr> ::= <line> |
           <expr> <line>
<line> ::= ifelse food-ahead [ <expr> ][ <expr> ] |
           <op>
<op>   ::= turn-left |
           turn-right |
           move
```

Listing 5.2: BNF-Koza grammar definition for the artificial ant problem.

```
N = {code, line, condition, op}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = code

P:
<code>      ::= <line> |
                <code> <line>
<line>      ::= <condition> |
                <op>
<condition> ::= ifelse food-ahead [ <line> ][ <line> ]
<op>        ::= turn-left |
                turn-right |
                move
```

Listing 5.3: BNF-O'Neill grammar definition for the artificial ant problem.

The goals of the experiments were to compare the results given by these two grammars in order to confirm or refute the conclusions of Robilliard, et al. (2006), and highlight the

impact of these grammars on the performance of the Grammatical Evolution algorithm (the success rate over a number of evolutionary runs) and the quality of the solutions it produces (the number of moves required for each solution).

The difference between these grammars, as noted by Robilliard, et al. (2006), is that the BNF-O'Neill grammar does not allow multiple <*op*> statements or sequences of <*op*> and <*condition*> statements in the branches of the <*condition*> production rule as the BNF-Koza grammar allows in the corresponding rule (first production rule of <*line*> non terminal). The BNF-O'Neill grammar just allows either one condition statement or one operator in each branch of the *ifelse* condition statement.

A tableau – with a style similar to O'Neill and Ryan (2001; 2003) of summarising information – that describes the problem and the experiments configuration can be seen in Table 5.2. Ten experiments were conducted (5 using BNF-Koza and 5 using BNF-O'Neill). Each experiment performed 100 evolutionary runs.

Table 5.2: Grammatical Evolution tableau for the Santa Fe Trail problem.

Objective	Find a computer program in the NetLogo programming language to control an artificial ant so that it can find all 89 pieces of food located on the Santa Fe Trail.
Terminal Operators	*turn-left*, *turn-right*, *move*, *food-ahead*.
Terminal Operands	None.
Fitness Cases	One fitness case.
Raw Fitness	Number of pieces of food eaten before the ant times out with 650 operations.
Wrapper	None.
BNF Grammar	Two different BNF grammar definitions are used (one for each set of experiments): BNF-Koza and BNF-O'Neill.
Evolutionary Algorithm	Steady-State Genetic Algorithm, Generation Gap = 0.9, Selection Mechanism: Roulette-Wheel Selection.
Initial Population	Randomly created with the following restrictions during the generation: Minimum Codons = 15 and Maximum Codons = 25
Parameters	Population Size = 500, Maximum Generations = 50 (without counting generation 0), Probability Mutation = 0.01, Probability Crossover = 0.9, Probability Duplication = 0.0, Probability Pruning = 0.0, Codon Size = 8, Wraps Limit = 10.

The configuration for all of the conducted experiments was exactly the same. Standard and widely used values from Evolutionary Computation (De Jong, 2006; Fogel, 2006; Ghanea-Hercock, 2003) and Grammatical Evolution (O'Neill and Ryan, 2001; 2003, pp.55-56) literatures were used without any attempt of optimising the configuration in advance for one BNF grammar definition or another. That is, the main objective of these experiments was to compare the two BNF grammar definitions in question, and to reveal an indicative performance of Grammatical Evolution using these grammars with a configuration that is similar to that used in the GE literature.

Note that the two operators introduced by Grammatical Evolution, pruning and duplication, are not used because either their usefulness is questioned or not justified. The first has been already discarded (O'Neill and Ryan, 1999d; 2001; 2003) and the second seems to be no longer used in the recent GE literature (O'Neill and Brabazon, 2008; Hemberg, O'Neill and Brabazon, 2008; Dempsey, O'Neill and Brabazon 2009, p.132; p.145; Hugosson, Hemberg, Brabazon and O'Neill, 2010). Note also, that the maximum allowable steps for the ant are set to 650, instead of 600 as in the GP literature or 615 as in the GE literature. The reason for this increase was to give Grammatical Evolution the chance to find more solutions using the investigated BNF grammars in order that the sample of the solutions found and the comparison of the effects of these grammars on the performance of Grammatical Evolution being more encompassing.

5.5 Results and Discussion

In a series of 100 evolutionary runs, which is conducted in each experiment, it is expected that some or many of these runs will be successful. Namely, they will terminate because they found a solution (an ant with raw fitness 89). In order to identify the best of all the solutions found in a series of 100 independent runs (namely of each experiment), a lexicographic ordering is applied. A lexicographic ordering can also be used to drive selection during an evolutionary run (Luke and Wiegand, 2002; Gagné, Schoenauer, Parizeau and Tomassini, 2006). It is clarified here that it is not the case in the experiments conducted and presented in this work (this approach is used here only to identify the best of the solutions found in a series of independent runs and not to drive selection during an evolutionary run).

Therefore, in the conducted experiments, the best solution of each experiment is calculated according to the following criteria: a) Raw Fitness (a higher value is considered better); b) Steps (a lower value is considered better); c) Phenotype length (a lower value is considered better); and d) Genotype length (a lower value is considered better).

If two or more solutions have the same value in one criterion then the next is checked and so on. Namely, comparing two solutions (therefore, with the same raw fitness 89), the solution requiring fewer steps is considered as the best. If they have the same number of required steps, then the solution with the smallest phenotype is considered as the best. If they have same values in both the steps and the phenotype length, then the solution with the smallest genotype is considered as the best. If all four criteria are exactly the same, then the best solution is identified randomly.

Table 5.3 and the Table 5.4 show the results of the experiments for each BNF grammar definition. The explanation of the fields in the tables is as follows:

a. Steps: This is the required steps (ant moves) of the best solution found in the particular experiment (over a series of 100 evolutionary runs).
b. Phenotype Length: The length in characters (including empty spaces) of the phenotype of the best solution found in the particular experiment.
c. Genotype Length: The length in bits of the genotype of the best solution found in the particular experiment.
d. Success Rate: This is how many evolutionary runs (percentage) found a solution (raw fitness 89).
e. Best: This is the best of the values in the table.
f. Avg.: This is the average of the values in the table.

Table 5.3: Results of GE using the BNF-Koza grammar definition.

	Exp #1	**Exp #2**	**Exp #3**	**Exp #4**	**Exp #5**	**Best**	**Avg.**
Steps	419	507	415	541	479	**415**	**472**
Phenotype Length	171	692	286	317	205	**171**	**334**
Genotype Length	1194	374	1066	1778	315	**315**	**945**
Success Rate	8%	11%	10%	6%	13%	**13%**	**10%**

Table 5.4: Results of GE using the BNF-O'Neill grammar definition.

	Exp #6	Exp #7	Exp #8	Exp #9	Exp#10	Best	Avg.
Steps	609	609	607	609	607	**607**	**608**
Phenotype Length	195	372	725	379	534	**195**	**441**
Genotype Length	427	355	1058	1209	391	**355**	**688**
Success Rate	80%	76%	75%	81%	74%	**81%**	**78%**

Figure 5.3 depicts the cumulative frequency of success measures over 500 evolutionary runs of the BNF-Koza grammar definition versus the BNF-O'Neill grammar definition.

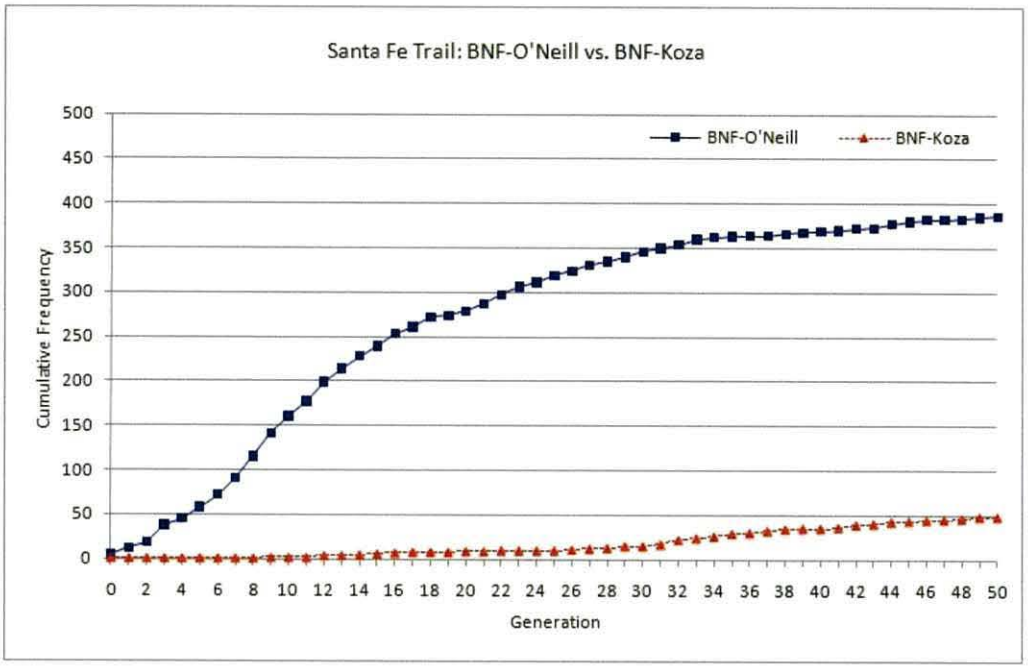


Figure 5.3: Cumulative frequency of success measures over 500 evolutionary runs.

Some obvious observations from the results are the following. The BNF-O'Neill grammar definition has a better success rate, approximately 80%. However, it produces the worst solutions in terms of efficiency (required steps), requiring each ant to perform approximately 610 moves. The BNF-Koza has a lower success rate, approximately 10%. In contrast, it produced the most efficient solutions within a range of 415 and 541 moves.

The best solution in terms of steps is given by the BNF-Koza grammar, requiring only 415 moves (Listing 5.4).

```
ifelse food-ahead
  [move]
  [ifelse food-ahead
```

```
    [turn-right]
    [turn-left
     ifelse food-ahead
       [move move
        turn-left
        turn-right]
       [turn-right]
     turn-right
     ifelse food-ahead
       [move]
       [ifelse food-ahead
          [turn-right]
          [turn-left]
          move
  ] ]  ]
```

Listing 5.4: Best SFT solution found by GE using BNF-Koza (415 steps).

Another excellent solution, also found again by the BNF-Koza grammar, requires only 419 moves. This is the solution with the shortest phenotype, found in all ten experiments conducted with the two grammars (Listing 5.5).

```
ifelse food-ahead
  [move]
  [turn-right
   ifelse food-ahead
     [turn-left]
     [turn-left turn-left]
   ifelse food-ahead
     [move]
     [turn-right]
   move
  ]
```

Listing 5.5: Another SFT solution of GE using BNF-Koza (419 steps).

Generally, the BNF-Koza grammar gives solutions with smaller phenotype size. In contrast, the solutions found with the BNF-O'Neill grammar have usually a very large and complex phenotype.

The above experimental results support the claim of Robilliard, et al. (2006) that the original Grammatical Evolution and Genetic Programming search spaces are not semantically equivalent, and therefore the comparison made by O'Neill and Ryan (2001; 2003, pp.55-58) is called into question. These experimental results cast doubt on the fairness of the comparison of Grammatical Evolution and Genetic Programming in O'Neill and Ryan (2001) and furthermore, whether Grammatical Evolution is a better approach than Genetic Programming in terms at least of the quality of the solutions. (The

BNF-O'Neill grammar produces solutions requiring many more steps than the solutions found with the BNF-Koza grammar). The BNF-Koza grammar has a larger search space than the BNF-O'Neill grammar. For this reason, it has a lower success rate but it gives better solutions (with fewer steps).

Also, it is observed that the success rates, as shown in the above results, of Grammatical Evolution using the BNF-O'Neill and the BNF-Koza grammars are close, as expected, to the success rates given in O'Neill and Ryan (2001; 2003, pp.57-58), for Grammatical Evolution and Genetic Programming (without using the solution length constrain) respectively, even though slightly different experimental setups and configurations were used. Namely, it appears in the corresponding figures of O'Neill and Ryan (2001; 2003, pp.57-58) that the first has a success rate of approximately 90% and the second of approximately 15%. This strengthens the validity of the experimental results presented here, in terms of reliability of the particular Grammatical Evolution and Santa Fe Trail implementations in Java and NetLogo (jGE, jGE NetLogo extension, and NetLogo models).

5.6 Further Investigation

Based on the insights gained from the results of the previous section, a series of new experiments have been conducted with the same configuration (see Table 5.2), but with or without wrapping, and using fixed length instead of variable length genomes as the standard GA does (Holland, 1975). The purpose of these experiments was firstly to investigate further the bias enforced by the BNF-O'Neill grammar definition, and secondly, to investigate the performance of Grammatical Evolution reconfigured with the absence of two of its key features: variable-length genomes and wrapping. Table 5.5, Table 5.6, Table 5.7, and Table 5.8 list the results.

The rows in these tables are explained as follows. "Runs" is the number of evolutionary runs performed in each experiment. "Codons" is the fixed-length size of the genome of the individuals (ants), measured in codons (8 bits strings). "Length (bits)" is the fixed-length size of the genome of the individuals (ants), measured in bits. "Steps" is the required steps (ant moves) of the best solution found in the particular experiment. And "Success Rate" is how many evolutionary runs (percentage) found a solution (ants with raw fitness 89).

Table 5.5: Results using the BNF-Koza grammar definition with fixed-length genomes and wrapping.

	Exp #1	Exp #2	Exp #3	Exp #4	Exp #5	Exp #6	Exp #7	Exp #8
Runs	100	100	100	100	100	100	100	100
Codons	15	25	50	75	100	150	300	500
Length (bits)	120	200	400	600	800	1200	2400	4000
Steps	650	547	385	405	441	377	425	471
Success Rate	**0%**	**7%**	**13%**	**13%**	**12%**	**14%**	**11%**	**15%**

Table 5.6: Results using the BNF-Koza grammar definition with fixed-length genomes without wrapping.

	Exp#9	Exp#10	Exp#11	Exp#12	Exp#13	Exp#14	Exp#15	Exp#16
Runs	100	100	100	100	100	100	100	100
Codons	15	25	50	75	100	150	300	500
Length (bits)	120	200	400	600	800	1200	2400	4000
Steps	650	650	405	547	395	385	405	447
Success Rate	**0%**	**0%**	**11%**	**6%**	**15%**	**16%**	**17%**	**10%**

Table 5.7: Results using the BNF-O'Neill grammar definition with fixed-length genomes and wrapping.

	Exp#17	Exp#18	Exp#19	Exp#20	Exp#21	Exp#22	Exp#23	Exp#24
Runs	100	100	100	100	100	100	100	100
Codons	15	25	50	75	100	150	300	500
Length (bits)	120	200	400	600	800	1200	2400	4000
Steps	613	609	609	607	607	607	611	607
Success Rate	**57%**	**43%**	**74%**	**59%**	**67%**	**60%**	**60%**	**59%**

Table 5.8: Results using the BNF-O'Neill grammar definition with fixed-length genomes without wrapping.

	Exp#25	Exp#26	Exp#27	Exp#28	Exp#29	Exp#30	Exp#31	Exp#32
Runs	100	100	100	100	100	100	100	100
Codons	15	25	50	75	100	150	300	500
Length (bits)	120	200	400	600	800	1200	2400	4000
Steps	650	609	607	607	607	607	607	607
Success Rate	**0%**	**21%**	**42%**	**41%**	**51%**	**51%**	**65%**	**67%**

The results confirm again that GE using BNF-O'Neill outperforms GE using BNF-Koza regarding the success rate, due to the fact that their grammars define different search spaces. Also, the results show that with fixed-length genomes the wrapping operator has a negative effect on the performance of Grammatical Evolution when the codon size range is between 100 and 300 for BNF-Koza, and above 300 for BNF-O'Neill. This is just an initial observation and further investigation is required before any conclusion is possible. Another observation is that GE using BNF-Koza gives generally better results when fixed-length genomes are used instead of variable length.

Both BNF grammar definitions have been furthermore compared using the same configuration as with the previous experiments, but with random search instead of the steady-state GA as the search engine component of GE, without wrapping, and with fixed-length genomes of 500 codons, in order to get stronger evidence that the BNF-O'Neill grammar defines a different and much smaller search space than the BNF-Koza grammar. This is confirmed with the results listed in Table 5.9. Also, the results show that GE using BNF-O'Neill does not perform much better than GE with random search using the same grammar definition. This is not the case with GE using BNF-Koza which outperforms GE with random search and using the same grammar definition.

It is recalled here that Grammatical Evolution using fixed-length genomes of 500 codons without wrapping has a success rate 10% with BNF-Koza and 67% with BNF-O'Neill.

Table 5.9: Results using standard GE with random search (as the search engine) and without wrapping.

	BNF-Koza (Exp #33)	BNF-O'Neill (Exp #34)
Runs	1000	1000
Codons	500	500
Length (bits)	4000	4000
Steps	527	607
Success Rate	**1.4%**	**50.2%**

Note that a pure random search was implemented and used as the search engine of GE in these experiments. Namely, during the GE evolutionary run, a new generation was created randomly replacing the parent population, instead of using parents selection, recombination, mutation, and replacement as happens in standard GE. For the creation of

the random offspring populations, no special methods were used and no constraints were enforced except the fixed-length genotype size.

One more reason for conducting the experiments described in this section – except the investigation of the two grammars – was to investigate whether there are better solutions for the Santa Fe Trail Problem than those found in the experiments described in the previous section and those mentioned in the Genetic Programming and Grammatical Evolution literature. Koza's best solution (1992, p.154) requires 545 steps. O'Neill and Ryan's (2003, p.56) best mentioned solution requires 615 steps. In these new experiments, a total of 5200 evolutionary runs (2600 for each grammar definition) with various search engines (fixed-length GE with/without wrapping, and Random Search) have been executed.

What was found is that there are better solutions (found using Grammatical Evolution with the BNF-Koza grammar) and that no solution has been produced with the BNF-O'Neill grammar with fewer than 607 steps. The best solution found with the BNF-Koza grammar requires only 377 steps (Listing 5.6). Grammatical Evolution found this solution using fixed-length genomes.

```
ifelse food-ahead
  [move]
  [ifelse food-ahead
    [turn-right]
    [ifelse food-ahead
      [turn-left turn-right]
      [turn-right]
    ]
   ifelse food-ahead
     [move]
     [ifelse food-ahead
       [turn-left move move turn-right]
       [turn-left turn-left
        ifelse food-ahead
          [move move]
          [turn-right]
       ]
      move
     ]
  ]
```

Listing 5.6: Best SFT solution found by fixed-length GE (steps 377).

The best solution found using the BNF-O'Neill grammar definition requires 607 moves (Listing 5.7).

```
ifelse food-ahead
  [move]
  [turn-left]
ifelse food-ahead
  [move]
  [turn-left]
move
turn-left
ifelse food-ahead
  [ifelse food-ahead
    [turn-right]
    [turn-left]
  ]
  [turn-left]
```

Listing 5.7: Best SFT solution found using BNF-O'Neill (607 steps).

The experimental results have confirmed that the BNF-O'Neill grammar definition (the standard grammar definition used in the GE literature) biases the search space in such a way that it selects a portion of the search space where no better solutions (with fewer steps than 607) could be, easily at least, found. The reduction of the search space may also explain the much better success rate (solutions found) of Grammatical Evolution against Genetic Programming. Consequently, by focusing the search in a narrower area of the search space, Grammatical Evolution (using the BNF-O'Neill grammar definition) performs better than Genetic Programming (using a search space semantically equivalent with that of the BNF-Koza grammar definition). This is in terms of effectiveness (solutions found), but worse than Genetic Programming in terms of efficiency (solution quality – required steps).

By comparing the success rates for random search using BNF-Koza and BNF-O'Neill grammar definitions as shown in Table 5.9, it is obvious from their huge difference (1.4% and 50.2% respectively) that the bias enforced by the BNF-O'Neill grammar is very significant when compared with that of BNF-Koza. This gives an unfair advantage to Grammatical Evolution using BNF-O'Neill when comparing it with Genetic Programming.

Finally, the observation that Grammatical Evolution using the BNF-O'Neill grammar definition did not find in the experiments that were conducted any solution with less than

607 steps, raised the obvious question whether Grammatical Evolution using the search space defined by BNF-O'Neill is not able to find solutions requiring less than 607 steps or just it was easier for Grammatical Evolution to find solutions requiring more than 606 steps and consequently the search was just stopped in that point, due to termination of the evolutionary run because a solution has been found.

For this reason, a new series of experiments was conducted using the same parameters as in the experiments described in the previous section (see Table 5.2) except the "Ant Max Steps" limit which from 650 has been set to 606. In particular, 5 experiments were conducted of 100 evolutionary runs each (a total of 500 evolutionary runs). The results of these experiments are shown in Table 5.10.

Table 5.10: Results using the BNF-O'Neill grammar definition and maximum ant steps limit 606.

	Exp#35	Exp#36	Exp#37	Exp#38	Exp#39	Best
Runs	100	100	100	100	100	**100**
Fitness	88	88	88	88	88	**88**
Steps	606	606	606	606	606	**606**
Success Rate	0%	0%	0%	0%	0%	**0%**
Avg. Success Rate	**0%**					

The rows in this table are explained as follows. "Runs" is the number of evolutionary runs performed in the experiment. "Fitness" is the raw fitness of the best ant found in the particular experiment. "Steps" is the required steps (ant moves) of the best ant found in the particular experiment. "Success Rate" is how many evolutionary runs (percentage) found a solution. And "Avg. Success Rate" is the average success rate of the five experiments.

These results confirm that Grammatical Evolution using BNF-O'Neill and the parameter settings of Table 5.2 is not able to find solutions requiring 606 steps or less. Consequently, it is not able to solve the original Santa Fe Trail problem where a 600 steps limit is imposed (Koza 1992, p.150; Langdon and Poli, 1998a; 1998b; Robilliard, et al., 2006).

5.7 Conclusions

Grammatical Evolution literature (O'Neill and Ryan, 2001; O'Neill and Ryan 2003, p.55) uses in its benchmark against Genetic Programming in the Santa Fe Trail problem, a BNF grammar named here BNF-O'Neill, which restricts the original search space and excludes good solutions. The experiments of this chapter confirm the experimental results of Robilliard, et al. (2006) by showing that Grammatical Evolution gives less competitive results (in terms of success rate) on the Santa Fe Trail problem when the original search space defined by Koza (1992) is used, namely the BNF-Koza grammar definition. This casts doubt on the claim that Grammatical Evolution outperforms Genetic Programming in this problem (O'Neill and Ryan, 2001).

Furthermore, it has been shown that Grammatical Evolution is not capable of finding solutions with less than 607 steps when the biased search space defined by BNF-O'Neill grammar is used. Consequently it is not able to solve the Santa Fe Trail problem using this grammar when the maximum steps limit is set to 600 as in the original problem defined by Koza (1992).

The above findings raise the question whether Grammatical Evolution can be improved, using some form of language and/or search bias, so that it will be able to find efficient solutions (namely, requiring fewer steps) displaying at the same time a high success rate. Utilising generally applicable knowledge of a problem domain (not just of a problem instance) to bias the search, will this lead to a performance increase reducing at the same time the risk of excluding good solutions?

Additionally, further performance improvement could be achieved by reducing the impact of destructive crossover events (O'Neill, Ryan, Keijzer and Cattolico, 2003; Harper and Blair, 2005; 2006a; Hemberg, 2010) and bloating (Harper and Blair, 2006b) which are generally regarded to be GP and GE issues (O'Neill, Ryan, Keijzer and Cattolico, 2003) as discussed in section 2.2.8.

Chapter 6

Grammatical Bias Effects on the Santa Fe Trail

6.1 Introduction

The experimental results of the previous chapter demonstrate that the BNF-O'Neill grammar definition (Listing 5.3) states a declarative language bias that narrows the possible representations that the system can consider, increasing in this way the effectiveness in terms of success rate (percentage of solutions found) with the side effect of decreasing the efficiency in terms of the solution quality (the required steps by the artificial ant control program to solve the problem) because of the exclusion of areas of the search space where better solutions exist.

These results stimulated further research and investigation on the effects of bias – which is enforced through the grammar – in the performance of Grammatical Evolution on the Santa Fe Trail problem and on whether a bias can be enforced that increases at the same time both the effectiveness (success rate) and efficiency (solution quality) with or without changing the semantics of the original search space defined by BNF-Koza (Listing 5.2). Namely, by using a search bias with or without enforcing a language bias. The next section provides a background in grammatical bias and is followed by the presentation of the experiments that are conducted using Grammatical Evolution with a variety of biased grammars on the Santa Fe Trail problem. The experimental results provide useful insights about the effects in the performance of Grammatical Evolution of various forms of grammatical bias which are implemented through the incorporation of building blocks and knowledge encoding (about the structure of possible solutions) in the grammar.

6.2 Grammatical Bias

Whigham (1996) defines bias as "the factors that influence a learning system to favour certain hypotheses or strategies" and highlights the issues of typing, program structure and

inductive bias in Genetic Programming to show the need for declarative biasing with evolutionary learning techniques, namely representing knowledge about the problem in question explicitly given by the user of the system. He claims that "for GP to be truly applicable over a wide variety of problems, explicit language and search bias is necessary to restrict the search space and make the discovery of a suitable computer program tractable". Whigham (1995a; 1995b) was one of the first to propose a grammar-based GP system (McKay, et al., 2010) by using a context-free grammar to specify structures in the hypothesis language.

There are three major kinds of bias (Whigham, 1996), based on the description language and the learning algorithm: *selective bias*, *language bias*, and *search bias*. *Selective bias* enables two or more equivalent hypotheses (in terms of performance) to be distinguished, *language bias* restricts the possible hypotheses that can be constructed, and *search bias* refers to the factors that control the transformation of one hypothesis into another. Regarding the *search bias*, it is possible to change the search space structure – its connectivity and overall fitness landscape – without changing the space itself. Thus it is possible to use a change of grammar to alter the search bias, not only the language bias (McKay, et al., 2010). Robilliard, et al. (2006) use the term *representation bias* for search bias and they characterise two search spaces as *semantically equivalent* when every program that can be found in one search space can also be found in the other and vice versa. According to their definition, semantically equivalent spaces can be defined by grammars which state the same language bias regardless if they state the same or different search bias.

An important component of bias is *correctness* (Whigham, 1995b) which describes how well a bias is suited to a problem. Namely, if a bias is not correct, the solution to the problem cannot be expressed. Hence there is a trade-off between limiting the search space and discounting meaningful solutions. In the extreme case of designing a grammar to restrict the search space and speed up the search, the search space can be too heavily constrained excluding the solution from the language and making the problem impossible to solve (Murphy, 2011). Also, Banzhaf (1994) notes in a different context (constrained optimisation problems) that hard constraint might lead to solutions that are not optimized. McKay, et al. (2010) mentions that the declarative search space restriction is perhaps the most obvious benefit of using grammars in GP in order to reduce the search cost to find a

solution but with the concomitant risk that the solution may not be within the defined search space or perhaps more insidious that the solution may be isolated by the grammar constraints and may be difficult to search. If the grammar is wrongly designed and chosen, except that the solution may not lie within the defined search space, the constrained nature of the search space can render search more difficult. Namely, the grammar space is generally sparser than the corresponding expression-tree space, thus neighbourhoods may be sparser, and they may be less connected making it probable that the solution is difficult to reach (McKay, et al., 2010).

6.2.1 Modularity

Modularity in a BNF-based GP system enforces a bias in the learning system which, depending on the way modularity is implemented, can result in a type of language bias (e.g. replacing or removing existing productions) or in a type of search bias (e.g. addition of new productions or changing the selection probability of a production). Rosca and Ballard (1994) show that *Adaptive Representations* can discover hierarchical representations while learning to solve a problem demonstrating performance improvement over standard GP approaches because the reusable nature of the discovered functions changes the search space of the problem, resulting in a modified language and search bias.

Hemberg (2010) explores the grammar in Grammatical Evolution and studies several different constructions of grammars and operators for manipulating the grammars and the evolutionary algorithm demonstrating that representations have a strong impact on the efficiency of search. Indeed, he verifies the benefits in the solution of the artificial ant problem by Grammatical Evolution from a representation which states a grammatical bias towards modules (e.g. ADFs or building block structures).

On the other side, Gariday (2008) investigates the effect of modularity on search and shows that even though modularity improves in general the performance of evolutionary algorithms it may crowd and complicate, if left without guidance, the space structure resulting in a harder space. Also, he shows that creating high value modules or low value modules has a direct and decisive impact on performance.

The experiments of section 6.4 provide insights about the use of building blocks in the Santa Fe Trail problem that are implemented in the grammar as additions of productions consisting only of terminal symbols.

6.2.2 Knowledge Incorporation

McKay, et al. (2010) note that the ability to encode knowledge about the problem is one of the strongest justifications for the use of grammar-based GP and that it is enough to encode meta-knowledge about the solution space using context-free grammar languages. The commonest mode of operation to incorporate knowledge and narrow the search space is controlling the language bias imposed by the grammar (McKay, et al., 2010). Another way of knowledge encoding in the grammar is with changing the search space structure without restricting the search space (McKay, et al., 2010).

Furthermore, it is noted by Whigham (1996) that the *No Free Lunch Theorem* can be viewed as another argument for the use of bias with a general learning system. Namely, for a generic learning system to perform well over a broad range of problems, it must be able to incorporate knowledge about the problem domain.

Whigham (1995a; 1995b; 1996) examines the effect of applying bias in the grammar and shows in the 6-Multiplexer problem that using the *if* function as the first function in the program with the first argument being an address line – a form of language bias that represent the underlying form of the problem – results in a significant improvement over using an unbiased grammar. A similar approach is followed in the experiments of section 6.5.

6.3 Experimental Setup

A tableau – with a style similar to O'Neill and Ryan (2001; 2003) of summarising information – that describes the problem and the experimental configuration can be seen in Table 6.1. The settings are the same (except the grammar) with these used in the experiments of section 5.4 (Table 5.2) for the benchmarking of Grammatical Evolution using BNF-Koza on the Santa Fe Trail problem.

Table 6.1: Grammatical Evolution tableau for the Santa Fe Trail problem.

Objective	Find a computer program in the NetLogo programming language to control an artificial ant so that it can find all 89 pieces of food located on the Santa Fe Trail.
Terminal Operators	*turn-left*, *turn-right*, *move*, *food-ahead*.
Terminal Operands	None.
Fitness Cases	One fitness case.
Raw Fitness	Number of pieces of food eaten before the ant times out with 650 operations.
Wrapper	None.
BNF Grammar	Biased variations of the BNF-Koza grammar definition.
Evolutionary Algorithm	Steady-State Genetic Algorithm, Generation Gap = 0.9, Selection Mechanism: Roulette-Wheel Selection.
Initial Population	Randomly created with the following restrictions during the generation: Minimum Codons = 15 and Maximum Codons = 25.
Parameters	Population Size = 500, Maximum Generations = 50 (without counting generation 0), Probability Mutation = 0.01, Probability Crossover = 0.9, Probability Duplication = 0.0, Probability Pruning = 0.0, Codon Size = 8, Wraps Limit = 10.

The BNF-Koza grammar definition is shown in Listing 6.1. This is the grammar definition on which the biased grammar variations of the experiments conducted in this chapter are based.

```
N = {expr, line, op}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = expr

P:
<expr> ::= <line> |
           <expr> <line>
<line> ::= ifelse food-ahead [ <expr> ][ <expr> ] |
           <op>
<op>   ::= turn-left |
           turn-right |
           move
```

Listing 6.1: BNF-Koza grammar definition for the artificial ant problem.

In the following sections, the grammars which are used by Grammatical Evolution in the experiments are presented and described alongside with the experimental results for each of these grammars.

The objective of these experiments is to highlight the effect of each grammar in the performance of Grammatical Evolution in terms of problem solving effectiveness (success rate) and efficiency of solutions found (actions performed to complete). A primary purpose is to extract useful indications of whether the addition of particular building blocks (reusable code) as productions in the BNF grammar definition or the application of general applicable problem domain knowledge through the use of a bias toward conditional statements in the start non-terminal symbol of the grammar are approaches that merits further attention and utilisation.

6.4 Grammars with Building Blocks

The grammar definitions of the experiments conducted in this section augment the BNF-Koza grammar definition (Listing 6.1) by adding productions, consisting only of terminals, in the <*op*> non-terminal symbol. These productions can be considered as candidate building blocks, namely useful and reusable segments of code, which have been added in the grammar definition as indivisible terminals extending the set of the existing primitive action terminals *turn-left*, *turn-right*, and *move*.

The selection of the particular building blocks has been done by taking into account the specific characteristics of the Santa Fe Trail, namely the irregularities of the trail which in short are: *straight*, *corner*, *single gap*, *double gap*, *single gap at corner*, *double gap at corner* (short knight move), and *triple gap at corner* (long knight move). Therefore, these building blocks could be considered as high value modules.

6.4.1 Grammar Definitions

The presentation and description of the grammars is following. The grammar of Listing 6.2 (BB #1) adds in line (1) a conditional statement (building block) as a production rule in the non-terminal symbol <*op*>. This conditional statement solves the *straight* and *corner* parts of the trail. The chance that the <*op*> non-terminal is expanded to this conditional statement during the genotype-to-phenotype mapping is 25%.

```
N = {expr, line, op}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = expr
```

```
P:
<expr> ::= <line> |
           <expr> <line>
<line> ::= ifelse food-ahead [ <expr> ][ <expr> ] |
           <op>
<op>   ::= turn-left |
           turn-right |
           move |
           ifelse food-ahead [move][turn-right]          (1)
```

Listing 6.2: BNF-Koza with Building Blocks (BB) Version #1.

The grammar definition of Listing 6.3 (BB #2) adds two more conditional statements of the same type in lines (2) and (3) increasing the chance that the <*op*> non-terminal is expanded to this statement to 50%. Namely, it has been adjusted what Whigham (1996) calls *merit weighting* of a production (the probability of a production being selected).

```
N = {expr, line, op}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = expr

P:
<expr> ::= <line> |
           <expr> <line>
<line> ::= ifelse food-ahead [ <expr> ][ <expr> ] |
           <op>
<op>   ::= turn-left |
           turn-right |
           move |
           ifelse food-ahead [move][turn-right] |        (1)
           ifelse food-ahead [move][turn-right] |        (2)
           ifelse food-ahead [move][turn-right]          (3)
```

Listing 6.3: BNF-Koza with Building Blocks (BB) Version #2.

The bias toward the conditional statement, which has been added in the previous grammars, is stronger in the grammar definition of Listing 6.4 (BB #3) where it appears in total six times in lines (1) until (6) with the chance now approximately 65% that the <*op*> non-terminal is expanded to this building block.

```
N = {expr, line, op}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = expr

P:
<expr> ::= <line> |
           <expr> <line>
<line> ::= ifelse food-ahead [ <expr> ][ <expr> ] |
           <op>
```

```
<op>   ::= turn-left |
           turn-right |
           move |
           ifelse food-ahead [move][turn-right] |          (1)
           ifelse food-ahead [move][turn-right] |          (2)
           ifelse food-ahead [move][turn-right] |          (3)
           ifelse food-ahead [move][turn-right] |          (4)
           ifelse food-ahead [move][turn-right] |          (5)
           ifelse food-ahead [move][turn-right]            (6)
```

Listing 6.4: BNF-Koza with Building Blocks (BB) Version #3.

The grammar definition of Listing 6.5 (BB #4) enforces an even stronger bias toward the conditional statement, which appears in total nine times in lines (1) until (9) with the chance 75% that the <*op*> non-terminal is expanded to this type of production.

```
N = {expr, line, op}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = expr

P:
<expr> ::= <line> |
           <expr> <line>
<line> ::= ifelse food-ahead [ <expr> ][ <expr> ] |
           <op>
<op>   ::= turn-left |
           turn-right |
           move |
           ifelse food-ahead [move][turn-right] |          (1)
           ifelse food-ahead [move][turn-right] |          (2)
           ifelse food-ahead [move][turn-right] |          (3)
           ifelse food-ahead [move][turn-right] |          (4)
           ifelse food-ahead [move][turn-right] |          (5)
           ifelse food-ahead [move][turn-right] |          (6)
           ifelse food-ahead [move][turn-right] |          (7)
           ifelse food-ahead [move][turn-right] |          (8)
           ifelse food-ahead [move][turn-right]            (9)
```

Listing 6.5: BNF-Koza with Building Blocks (BB) Version #4.

The grammar definition of Listing 6.6 (BB #5) adds, to the primitive actions (*turn-left*, *turn-right*, and *move*) of the non-terminal symbol <*op*>, building blocks of code which solve all the irregularities of the trail. The production in line (1) solves the *straight* and *single gap* parts of the trail, the production in line (2) the *double gap*, the productions in lines (3) and (4) the *corner* and the *single gap at corner*, the productions in lines (5) and (6) the *double gap at corner*, and the productions in lines (7) and (8) the *triple gap at corner*.

```
N = {expr, line, op}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = expr

P:
<expr> ::= <line> |
           <expr> <line>
<line> ::= ifelse food-ahead [ <expr> ][ <expr> ] |
           <op>
<op>   ::= turn-left |
           turn-right |
           move |
           move  |                                          (1)
           move move |                                      (2)
           move turn-right |                                (3)
           move turn-left |                                 (4)
           move move turn-right |                           (5)
           move move turn-left |                            (6)
           move move move turn-right |                      (7)
           move move move turn-left                         (8)
```

Listing 6.6: BNF-Koza with Building Blocks (BB) Version #5.

The BNF grammar definition of Listing 6.7 (BB #6) extends the previous grammar in two ways. First, it wraps all the building blocks of Listing 6.6 (BB #5) in a conditional statement, see lines (5) – (12). Second, it adds four conditional statements which solve both the *straight* and *corner* parts more effectively. Namely, the productions in lines (1) and (3) solve effectively the *right corners* and the productions in lines (2) and (4) the *left corners*. They are added twice in order to double their chance to be selected against the other productions.

```
N = {expr, line, op}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = expr

P:
<expr> ::= <line> |
           <expr> <line>
<line> ::= ifelse food-ahead [ <expr> ][ <expr> ] |
           <op>
<op>   ::= turn-left |
           turn-right |
           move |
           ifelse food-ahead [move] [turn-right] |          (1)
           ifelse food-ahead [move] [turn-left] |           (2)
           ifelse food-ahead [move] [turn-right] |          (3)
           ifelse food-ahead [move] [turn-left] |           (4)
           ifelse food-ahead [move] [move] |                (5)
           ifelse food-ahead [move] [move move] |           (6)
           ifelse food-ahead [move] [move turn-left] |      (7)
```

```
            ifelse food-ahead [move] [move turn-right] |          (8)
            ifelse food-ahead [move] [move move turn-left] |      (9)
            ifelse food-ahead [move] [move move turn-right] |     (10)
            ifelse food-ahead [move] [move move move turn-left] | (11)
            ifelse food-ahead [move] [move move move turn-right]  (12)
```

Listing 6.7: BNF-Koza with Building Blocks (BB) Version #6.

Listing 6.8 (BB #7) adds to the grammar definition of Listing 6.6 (BB #5) a conditional statement in line (1) for providing a more general production than the productions of lines (2), (4) and (5), to solve both the *straight* and *corner* parts of the trail.

```
N = {expr, line, op}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = expr

P:
<expr> ::= <line> |
           <expr> <line>
<line> ::= ifelse food-ahead [ <expr> ][ <expr> ] |
           <op>
<op>   ::= turn-left |
           turn-right |
           move |
           ifelse food-ahead [move][turn-right]          (1)
           move  |                                       (2)
           move move |                                   (3)
           move turn-right |                             (4)
           move turn-left |                              (5)
           move move turn-right |                        (6)
           move move turn-left |                         (7)
           move move move turn-right |                   (8)
           move move move turn-left                      (9)
```

Listing 6.8: BNF-Koza with Building Blocks (BB) Version #7.

Finally, the grammar definition of Listing 6.9 (BB #8) removes from the original definition (BNF-Koza) the *turn-left* and *turn-right* terminals and replaces them with a conditional statement which solves the *straight* and *corner* parts of the trail.

```
N = {expr, line, op}
T = {turn-right, move, ifelse, food-ahead, [, ]}
S = expr

P:
<expr> ::= <line> |
           <expr> <line>
<line> ::= ifelse food-ahead [ <expr> ][ <expr> ] |
           <op>
<op>   ::= move |                                        (1)
           ifelse food-ahead [move][turn-right]          (2)
```

Listing 6.9: BNF-Koza with Building Blocks (BB) Version #8.

6.4.2 Experimental Results

For each of the above grammar definitions an experiment of 100 evolutionary runs is conducted. The Table 6.2 compares some properties of the setups and displays the experimental results. "Runs" is the number of evolutionary runs performed in each experiment, "Language Bias" is whether the grammar restricts the space of possible hypotheses, "Search Bias" is whether the grammar biases the search, "Building Blocks" is the number of the building blocks (code segments) added to the grammar, "IF-ELSE Blocks" is the number of conditional statement building blocks, "Steps" is the required steps of the best solution found in the particular experiment, "Genotype Size" is the size of the genotype in bits of the best solution found, and "Success Rate" is the percentage of successful runs (that found a solution).

Table 6.2: Results of GE using the BNF-Koza Building Blocks variations.

	BB #1	**BB #2**	**BB #3**	**BB #4**	**BB #5**	**BB #6**	**BB #7**	**BB #8**
Runs	100	100	100	100	100	100	100	100
Language Bias	No	No	No	No	No	No	No	Yes
Search Bias	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Building Blocks	1	3	6	9	8	12	9	1
IF-ELSE Blocks	1	3	6	9	0	12	1	1
Steps	447	407	397	397	--	--	541	385
Genotype Size	1458	2468	460	11629	--	--	1060	6700
Success Rate	**19%**	**30%**	**30%**	**32%**	**0%**	**0%**	**3%**	**1%**

6.4.3 Discussion

All the grammars, except the last of Listing 6.9 (BB #8), define a search space semantically equivalent (same language bias) to that defined by the BNF-Koza grammar definition (Listing 6.1) but they state different search biases.

The BNF grammars BB #1 (19%), BB #2 (30%), BB #3 (30%), and BB #4 (32%) result in a higher success rate than BNF-Koza (10%, see Table 5.3). Instead, Grammatical Evolution using the other BNF grammars either could not find a solution (BB #5 and BB #6) or performed very poorly (BB #7 and BB #8).

The grammars that enabled Grammatical Evolution to perform well have all of them the building block *ifelse food-ahead [move][turn-right]* which will be called for shortness "basic condition". It is noted here that the "basic condition" solves the *straight* and *corner* parts which are the dominant parts of the trail. Instead, the grammars that have specialised building blocks that solve particular irregularities of the trail could not find a solution (BB #5 and BB #6) or had a very low success rate (BB #7) probably because they define a more difficult and complex search space due to the fact that the irregularities their building blocks solve are few in the trail in contrast to the relative high chance of these building blocks being selected from the <*op*> non-terminal against the primitive actions. The BB #7 grammar definition has additionally one "basic condition" building block and was able to find a solution, although with a very low success rate (3%).

Also, the absence of the terminals *turn-left* and *turn-right* in BB #8 resulted in a 1% success rate even though the "basic condition" substitutes the *turn-right* action with the same number of required steps and can substitute the *turn-left* action using three steps instead of one (by performing a 270^{o} rotation). Consequently, the absence of the *turn-left* terminal operator affects the economy of the required moves to find all pieces of food in the trail. This economy factor is the main reason for the ineffectiveness of this grammar in conjunction with the fact that the search space becomes more complex (rotations are required instead of a single turn).

An interesting observation from the experimental results is that BB #2 (30%), BB #3 (30%) and BB #4 (32%), which have building blocks only of the "basic condition" type, achieved similar success rates even though BB #3 has double and BB #4 triple number of useful "basic condition" blocks than BB #2. It seems that there is some kind of "equilibrium" regarding the impact of the number of these building blocks in the success rate, with three blocks to be the lower bound (the same number with the primitive actions in the grammar). This lower bound could be called here arbitrarily the "50% rule" (for the specific grammar, building blocks and problem) because the added building blocks are half of all action terminal symbols of the <*op*> non-terminal.

In order to investigate the above assumption, new experiments are required using grammars with more "basic condition" blocks. Also, if this observed "equilibrium" is true, the question that arises is whether it has an upper bound after which the performance decreases, and if yes, what is its value. For this reason, seven new experiments have been

conducted using the same GE experimental configuration with the grammars BB #1 until BB #4 with the only difference being the number of the added "basic condition" building blocks in the grammar. Figure 6.1 shows the experimental results (success rates) of the new setups where 12, 15, 18, 27, 36, 72, and 144 "basic condition" building blocks were added, and compares them with the results of the previous setups (BB #1 - BB #4) and the original BNF-Koza grammar (zero number of added blocks).

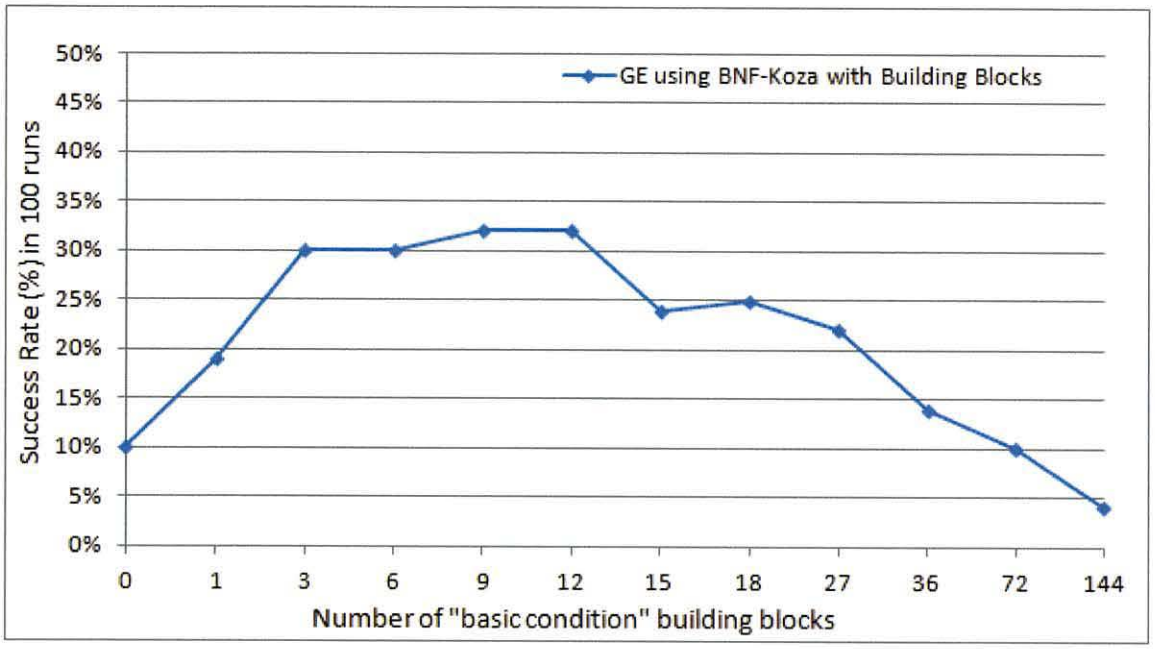


Figure 6.1: Success rate over 100 evolutionary runs of GE using BNF-Koza with and without a variety number of "basic condition" blocks additions in the grammar.

The graph shows that when adding one "basic condition" the success rate increases from 10% (GE using original BNF-Koza without building blocks) to 19%. The addition of three or more "basic condition" blocks results in a success rate approximately 30% until the point of the addition of fifteen blocks (24%). After that, a decrease in performance occurs continuing down to a success rate of 10% with 72 blocks, and finally of 4%, when the blocks are 144, which is lower than the success rate of BNF-Koza without using building blocks (10%).

The experimental results confirm the previously stated hypothesis that in this specific setup (problem, grammar and type of added building blocks) there is an "equilibrium" regarding the impact of the number of the added building blocks in the success rate, with lower and upper bounds three and twelve building blocks approximately.

Probably the reason of an upper bound, after which the success rate decreases and reaches finally a point of lower performance than without using building blocks, is the decrease of the probability the *turn-left* terminal being selected which proved to be important as shown in the results of BB #7. Therefore, as each terminal symbol, primitive (original) or complex (building block) has a merit (contribution) in the search for a solution, if the addition of more building blocks actually "precludes" a useful terminal (of which the merit cannot be substituted by an equivalent terminal or building block) by substantially reducing its probability of being selected, this has a negative impact in the performance. Also, there is some point after which the benefit of adding more useful building blocks will be lost due to the negative impact of lowering the probability of utilising the merit to the solution of some original (primitive) terminal symbol or other useful building block. This does not happen if the merit of the "precluded" terminal symbol is substituted by some added building blocks. For example, if the merit (contribution) of the *turn-left* terminal symbol is substituted with the addition of a new "basic condition" which makes the ant turn left instead of right, this would probably benefit the performance. This reveals the impact on the performance of the number of the added building blocks, especially if their usefulness and diversity is not known in advance or is not at least controllable.

Gariday (2008) provides a theoretical framework for studying the impact of modularity in the search space of evolutionary algorithms and an experimental study using Modular GA. Further investigation about the impact of the number and the type of the added building blocks in the performance of Grammatical Evolution could be beneficial to future research of GE applications of modularity.

6.5 Grammars with Conditional Statement Bias

The grammar definitions of the experiments conducted in this section declaratively bias the search space defined by the BNF-Koza grammar definition (Listing 6.1) by enforcing the creation of control programs which start with *if-else* conditional statements. Namely, the resulting ant control programs are forced to perform a check (condition) whether there is food ahead and if yes to take the appropriate action (behaviour). When no food is found ahead, the control program is forced to take some other action and/or perform a new check (alternative behaviour) and so on. This could be considered as an attempt to encode in the

grammar a form of domain knowledge for agent-oriented problems, which is generally applicable to such problems and not only to a specific problem instance.

The experiments test the hypothesis whether a bias toward *if-else* conditional statements in the start of the ant's control program will increase the performance of Grammatical Evolution in the Santa Fe Trail problem. The presentation and description of the grammars is following.

6.5.1 Grammar Definitions

In the grammar of Listing 6.10 (CB #1), a language bias has been applied resulting in a grammar which defines a different and restricted search space than that defined by BNF-Koza (Listing 6.1). It biases the language to constructs in which the control program performs successive checks whether there is food ahead. Note that the production of line (1) constructs consecutively nested checks of the form: *check*, if food found *take some action* else *take some action* and *check* again. The bias toward constructs with a conditional statement in the start of the program is 100% (therefore 0% toward a primitive action *turn-left*, *turn-right*, or *move*). Instead, the bias toward a conditional statement or a primitive action in BNF-Koza is 50%. Also, the *IF branch* of a conditional statement in grammar CB #1 is always (100%) a single primitive action.

```
N = {behaviour, op}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = behaviour

P:
<behaviour> ::= ifelse food-ahead [ <op> ][ <op> <behaviour> ] |          (1)
                ifelse food-ahead [ <op> ][ <op> ]                        (2)
<op>        ::= turn-left |
                turn-right |
                move
```

Listing 6.10: BNF-Koza with Conditional Bias (CB) Version #1.

The grammar of Listing 6.11 (CB #2) adds to the previous grammar the production rule <*op*> to the non-terminal symbol <*behaviour*> in line (1). This allows a <*behaviour*> non-terminal to be expanded to a primitive action (*turn-left*, *turn-right*, *move*) with 33% chance of being selected (against 0% in CB #1) and to a conditional statement with 67% chance of being selected (against 100% in CB #1) weakening in this way the language

bias. The <*op*> non-terminal symbol is always (100%) expanded to a single primitive action (*turn-left*, *turn-right*, *move*) as in the previous grammar.

```
N = {behaviour, op}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = behaviour

P:
<behaviour> ::= ifelse food-ahead [ <op> ][ <op> <behaviour> ] |
                ifelse food-ahead [ <op> ][ <op> ] |
                <op>                                                    (1)
<op>        ::= turn-left |
                turn-right |
                move
```

Listing 6.11: BNF-Koza with Conditional Bias (CB) Version #2.

Both grammars which are defined in Listing 6.10 (CB #1) and Listing 6.11 (CB #2) state a language bias. The remaining grammars in this section state only a search bias. Namely, they define a search space semantically equivalent to that defined by BNF-Koza (same language bias) but they aim to bias the search toward those areas of the search space which are defined by the previous grammars CB #1 and CB #2. This search bias is implemented in the grammar definition of Listing 6.12 (CB #3) with the "wrapping" of the BNF-Koza grammar definition (Listing 6.1) by the grammar of Listing 6.11 (CB #2). Namely, the non-terminal symbol <*op*> in line (1) of Listing 6.11 (CB #2) has been replaced by the non-terminal symbol <*expr*> of the BNF-Koza definition of Listing 6.1. In the grammar CB #3, the <*expr*> non-terminal symbol (equivalent to the <*op*> non-terminal symbol of the next grammars presented in this section) expands to conditional statements or primitive actions with chance of 50%. Therefore, the bias toward conditional statements in the start of the program in this grammar is approximately 83% (67% from the start symbol and half of the 33% from the <*expr*> non-terminal). The <*op*> non-terminal symbol of this grammar always expands (100%), as in the previous grammars, to a single primitive action.

```
N = {behaviour, op, expr, line}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = behaviour

P:
<behaviour> ::= ifelse food-ahead [ <op> ][ <op> <behaviour> ] |
                ifelse food-ahead [ <op> ][ <op> ] |
                <expr>                                                  (1)
```

```
<expr>      ::= <line> |                                                    (2)
                <expr> <line>                                               (3)
<line>      ::= ifelse food-ahead [ <expr> ][ <expr> ] |                    (4)
                <op>                                                        (5)
<op>        ::= turn-left |
                turn-right |
                move
```

Listing 6.12: BNF-Koza with Conditional Bias (CB) Version #3.

The grammar definition of Listing 6.13 (CB #4) is semantically equivalent (defines the same space of possible hypotheses) with BNF-Koza (Listing 6.1) and CB #3 (Listing 6.12) grammars but biases the search in a different way. Namely, it promotes the selection of primitive actions (*turn-left*, *turn-right*, *move*) when the non-terminal symbol of the third production <*op*> of the <*behaviour*> non-terminal symbol has to be expanded. This is done with the replacement of <*expr*> with the <*op*> non-terminal symbol in CB #4 which is expanded with a chance 3/8 (approximately 37%) directly to a single primitive action. Consequently, the bias toward conditional statements in the start of the program in this grammar is approximately 77%. Also, in this grammar the *IF branch* of the conditional statements of the start symbol does not expand anymore only to a single primitive action. The chance now of primitive actions being selected in these *IF branches* is approximately 70% (from 100% in the previous grammars).

```
N = {behaviour, op, complex-op, single-op, line}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = behaviour

P:
<behaviour>  ::= ifelse food-ahead [ <op> ][ <op> <behaviour> ] |
                 ifelse food-ahead [ <op> ][ <op> ] |
                 <op>                                                       (1)
<op>         ::= <single-op> |                                              (2)
                 <single-op> |                                              (3)
                 <single-op> |                                              (4)
                 <complex-op> |                                             (5)
                 <complex-op> |                                             (6)
                 <complex-op> |                                             (7)
                 <complex-op> |                                             (8)
                 <complex-op>                                               (9)
<complex-op> ::= <line> |                                                   (10)
                 <complex-op> <line>                                        (11)
<line>       ::= ifelse food-ahead [ <complex-op> ][ <complex-op> ] |       (12)
                 <single-op>                                                (13)
<single-op>  ::= turn-left |                                                (14)
                 turn-right |                                               (15)
                 move                                                       (16)
```

Listing 6.13: BNF-Koza with Conditional Bias (CB) Version #4.

In the grammar definition of Listing 6.14 (CB #5), the bias of the <*op*> non-terminal symbol toward primitive actions is increased to 75%. Therefore, the bias toward conditional statements in the start of the program in this grammar is approximately 74%.

```
N = {behaviour, op, complex-op, single-op, line}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = behaviour

P:
<behaviour>  ::= ifelse food-ahead [ <op> ][ <op> <behaviour> ] |
                 ifelse food-ahead [ <op> ][ <op> ] |
                 <op>
<op>         ::= <single-op> |
                 <single-op> |
                 <single-op> |
                 <complex-op> |                                          (1)
                 <complex-op> |                                          (2)
                 <complex-op>                                            (3)
<complex-op> ::= <line> |
                 <complex-op> <line>
<line>       ::= ifelse food-ahead [ <complex-op> ][ <complex-op> ] |
                 <single-op>
<single-op>  ::= turn-left |
                 turn-right |
                 move
```

Listing 6.14: BNF-Koza with Conditional Bias (CB) Version #5.

Finally, in the grammar of Listing 6.15 (CB #6), the bias of the <*op*> non-terminal symbol toward primitive actions is increased to 88%. Consequently, the bias toward conditional statements in the start of the program in this grammar is further decreased to approximately 70%.

```
N = {behaviour, op, complex-op, single-op, line}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = behaviour

P:
<behaviour>  ::= ifelse food-ahead [ <op> ][ <op> <behaviour> ] |
                 ifelse food-ahead [ <op> ][ <op> ] |
                 <op>
<op>         ::= <single-op> |
                 <single-op> |
                 <single-op> |
                 <complex-op>                                            (1)
<complex-op> ::= <line> |
                 <complex-op> <line>
<line>       ::= ifelse food-ahead [ <complex-op> ][ <complex-op> ] |
                 <single-op>
<single-op>  ::= turn-left |
                 turn-right |
                 move
```

Listing 6.15: BNF-Koza with Conditional Bias (CB) Version #6.

6.5.2 Experimental Results

Table 6.3 displays properties of the setups and the experimental results. The description of the fields of the results table is following: "Runs" is the number of evolutionary runs performed in each experiment, "Language Bias" is whether the grammar restricts the space of possible hypotheses, "Search Bias" is whether the grammar biases the search, "IF Statement %" is the chance of a conditional statement in the start of the program, "Action %" is the probability of a primitive action *(turn-right, turn-left, move)* in the start of the program, "Action % in IF Branch" is the bias toward primitive actions in the *IF branch* of a conditional statement of the start symbol, "Steps" is the required steps of the best solution found, "Genotype Size" is the size of the genotype in bits of the best solution found, and "Success Rate" is the percentage of successful runs.

Table 6.3: Results of GE using the BNF-Koza Conditional Bias variations.

	CB #1	CB #2	CB #3	CB #4	CB #5	CB #6
Runs	100	100	100	100	100	100
Language Bias	Yes	Yes	No	No	No	No
Search Bias	Yes	Yes	Yes	Yes	Yes	Yes
IF Statement %	100%	67%	83%	77%	74%	70%
Action %	0%	33%	17%	23%	26%	30%
Action % in IF Branch	100%	100%	100%	70%	75%	88%
Steps	405	405	405	397	395	395
Genotype Size	111	128	1028	2138	1223	472
Success Rate	**99%**	**93%**	**82%**	**19%**	**35%**	**74%**

6.5.3 Discussion

Grammatical Evolution using the grammars CB #1 and CB #2, which state a strong declarative language bias, achieves high success rates 99% and 93% respectively, higher than these with BNF-Koza (10%) or BNF-O'Neill (78%). This demonstrates the potentiality of this type of enforced bias, namely, toward conditional checks in the start of the control program of the type in Listing 6.10 and Listing 6.11.

The grammars CB #3, CB #4, CB #5, and CB #6 define search spaces semantically equivalent with this defined by BNF-Koza but they state different search biases. All these

grammars outperform BNF-Koza in terms of success rate. The grammar CB #3, which achieves a success rate of 82%, and the grammar CB #6, which achieves a success rate of 74%, define search spaces which are closer to these of CB #1 and CB #2 than these of CB #4 and CB #5, due to the high "Action % in IF Branch" rate.

An interesting observation is that even though CB #1, CB #2, and CB #3 result in significant high success rates, they found solutions requiring at least 405 steps, when it is known from the experiments conducted in Chapter 5 that better solutions exist (requiring less steps). This could be thought as a result of the strong declarative language bias which may exclude areas where better solutions exist (e.g. the solution of Listing 5.6 cannot be constructed by the grammars CB #1 and CB #2) or makes these solutions practically unreachable even though they exist in the biased search space (such a case could be CB #3 which defines a search space semantically equivalent to the original). It seems that the main reason for this performance regarding the solution quality is that the first branch of the conditional statements of the <*behaviour*> non-terminal symbol allows the expansion to only one primitive action (*turn-left*, *turn-right*, or *move*) in these grammars.

Instead, the grammars CB #4, CB #5, and CB #6 allow the expansion to all possible structures of both branches of the conditional statements of the <*behaviour*> non-terminal symbol. This results in lower success rates but in also finding better solutions (requiring for example 395 or 397 steps). It is also noticed that CB #6 has a strong search bias toward primitive actions in the first branch of the conditional statements of the <*behaviour*> non-terminal, which could explain its significantly higher success rate than those of the CB #4 and CB #5 grammars.

Another observation is that the grammars CB #4 (19%), CB #5 (35%), and CB #6 (74%) show an increase in performance which is inverse to the selection chance of the <*complex-op*> production rule of the <*op*> non-terminal symbol. It is noted that even though CB #6 achieves the higher success rate between these grammars, its bias seems to be similar to that enforced in CB #3 due to the high "Action % in IF Branch" value, which questions whether better solutions than this found in the conducted experiment are practically reachable using this grammar.

6.6 Conclusions

According to Whigham (1996), for Genetic Programming to be truly applicable over a wide variety of problems, explicit language and search bias is necessary to restrict the search space and make the discovery of a suitable computer program tractable. An important component of bias is *correctness* (Whigham, 1995b) which describes how well a bias is suited to a problem. Namely, if a bias is not correct, the solution to the problem cannot be expressed (Banzhaf, 1994; Whigham, 1996; Murphy 2011), or the constrained nature of the search space may render the search more difficult (McKay, et al., 2010).

The experiments of this chapter highlight the effects and demonstrate the effectiveness of two general forms of grammatical bias in the Santa Fe Trail problem. The first is an application of modularity using building blocks consisting of useful code segments implemented as additions of productions in the grammar. The second form of grammatical bias encodes in the grammar a generally applicable domain knowledge of the class of agent-oriented problems which is implemented as checks (conditional statements) in the start of the ant's control program.

It is shown that utilising these forms of grammatical bias a performance increase of Grammatical Evolution in the Santa Fe Trail problem in terms of both effectiveness (success rate) and efficiency (solution quality) can be achieved. Additionally, it is shown that a strong declarative language or search bias may result in a higher success rate but there is the risk that this may be achieved against the solution quality.

Chapter 7

Constituent Grammatical Evolution

7.1 Introduction

In this chapter, Constituent Grammatical Evolution (CGE) is presented, a new evolutionary automatic programming algorithm that extends the standard Grammatical Evolution algorithm by incorporating the concepts of constituent genes and conditional behaviour-switching. CGE builds from elementary and more complex building blocks a control program, which dictates the behaviour of an agent, and it is applicable to the class of problems where the subject of search is the behaviour of an agent in a given environment. It takes advantage of the powerful Grammatical Evolution feature of using a BNF grammar definition as a plug-in component to describe the output language to be produced by the system.

The main benchmark problem in which CGE is evaluated is the Santa Fe Trail problem. Furthermore, CGE is evaluated on three additional problems, the Los Altos Hills, the Hampton Court Maze, and the Chevening House Maze (these problems are presented in detail in sections 2.5.4 and 2.5.5). The experimental results demonstrate that Constituent Grammatical Evolution improves the standard Grammatical Evolution algorithm in all of these problems, in terms of both problem solving effectiveness (success rate) and efficiency of solutions found (actions performed to complete).

Constituent Grammatical Evolution, as well as part of the work presented in this chapter, has been first published in Georgiou and Teahan (2011).

7.2 Motivation and Main Concepts

The goal of Constituent Grammatical Evolution is to improve Grammatical Evolution in terms of effectiveness (percentage of solutions found in a total of evolutionary runs) and efficiency (solution quality in terms of the characteristics of the problem in question) in agent-oriented problems like the artificial ant. The main benchmark problem in which

CGE is first evaluated is the Santa Fe Trail problem. Therefore, the specific goals are: first, to improve the success rate of evolutionary runs in the Santa Fe Trail problem against standard Grammatical Evolution; and second, to find solutions which will require fewer steps than the solutions the Grammatical Evolution algorithm is usually able to find.

Constituent Grammatical Evolution tries to tackle three of the Grammatical Evolution issues discussed in section 2.2.8 and Chapter 5 of this text, in the following ways:

- restricting the search space using generally applicable domain knowledge of the problem in question without excluding good solutions;
- reducing the impact of destructive crossover events; and
- resolving the genotype bloating.

The above are achieved by augmenting and improving the standard Grammatical Evolution algorithm, with the incorporation of three unique features of the Constituent Grammatical Evolution algorithm:

- the conditional behaviour-switching approach, which biases the search space toward useful areas applying generally applicable knowledge of the agent-oriented problems domain (conditional checks);
- the incorporation of the notion of genes, which provides useful and reusable building blocks and reduces the impact of destructive crossover events through modularity; and
- the restriction of the genotype maximum size, which resolves the genotype bloat phenomenon.

The BNF based genotype-to-phenotype feature of Grammatical Evolution enables the implementation of two of the CGE's unique features (genes and behaviour-switching) in a straightforward and very flexible way through the modification and adaptation of the BNF grammar definition as is shown in the forthcoming sections.

7.2.1 The Constituent Genes Concept

The cost of exploration via crossover in Genetic Programming, therefore in Grammatical Evolution as well, is the possible destruction of building blocks, namely the disruption of

functional parts of the individual (O'Neill, Ryan, Keijzer and Cattolico, 2001). The impact of destructive crossover events in Grammatical Evolution can be reduced with a mechanism which indicates and preserves useful blocks of genetic material.

Angeline and Pollack (1993) note that modularity freezes possible useful genetic material, by protecting it from the destructive effect of genetic operators. Also, modularity helps in decreasing the average size of individuals (Rosca and Ballard, 1994). Furthermore, Hemberg (2010) examines disruptions in the phenotype caused by a change in genotype (e.g. crossover and mutation) in Grammatical Evolution. He shows that the fewer non-terminals there are in the grammar, the less susceptible it will be to disruption when the genotype changes. Also, he shows that probability for the phenotype to change increases with the position of the codon in the chromosome and that a longer chromosome makes disruptions more probable.

Therefore, indicating and preserving useful blocks of genetic material (namely useful functional parts) will benefit Grammatical Evolution regarding the impact of destructive crossover events in two ways. First, these building blocks will be protected from crossover because they will correspond to undivided segments of useful functional parts, and second they will shorten the chromosome of the individual (making disruptions less probable) because they will compress more information into a single choice of a production with fewer codon readings required to get to larger phenotypes. Additionally, if the building blocks are incorporated in the grammar in a way that does not increase the number of the non-terminal symbols, the resulted grammar will not become more susceptible to disruption during crossover than before the addition.

CGE implements modularity by taking inspiration from nature, in particular from *genes*. In biology, a gene is a sequence of nucleic acid base pairs that encodes a program with a specific function and controls characteristics of an organism (Mayr 2002, p.102). Genes are the basic unit of heredity and the set of the genes found in a population constitute the *gene pool* of this population (Mayr 2002, pp.100-103; p.116).

The concept of genes in CGE is implemented using *constituent genes* which are so named because they form basic elements for the construction of segments (building blocks) of the phenotype of an individual. The first step of the algorithm is the evaluation of randomly created candidate constituent genes and the selection of the fittest of them to form the

genes pool. Then the phenotypes of the constituent genes of the pool are added in the grammar definition of the problem in question as productions, and finally a Grammatical Evolution run starts using the modified grammar.

Namely, genotypes of candidate constituent genes are created randomly. They are mapped to their phenotypes using the original (unbiased) grammar of the problem in question, and then they are evaluated in randomly selected smaller parts of the problem. The best of them are selected for the genes pool according to their fitness value. The phenotypes of the constituent genes of the pool are added in the grammar definition as productions in a non-terminal symbol which expands to terminals semantically equivalent with the added phenotypes in order to not violate the syntax of the grammar. Therefore, the set of the primitive terminal symbols of the BNF grammar definition is enriched with the phenotypes of the constituent genes (in this context, as primitive terminals are defined as the smallest units that can form a valid construct of the language). The modified grammar is then used by Grammatical Evolution to perform an evolutionary run.

The exact implementation of the evaluation and selection of the constituent genes depends on the kind of the problem. For example, in the case of the artificial ant and maze search problems, the candidate constituent genes are evaluated by randomly "throwing" them in the environment and executing their control programs for a specified number of iterations to evaluate the degree they contribute toward the objective of the problem. A detailed description of the algorithm and a specific implementation in the artificial ant problem are provided in the following sections 7.3 and 7.4 respectively.

It is worthwhile to note here that randomness is intentionally chosen as the creation mechanism of the constituent genes of the CGE algorithm, in order to evaluate the usefulness of this concept without being interfered and affected by the positive and/or negative aspects of a probably more useful and well suited genes creation mechanism. One could think also about this random process as an analogy to the hypothesis of the accidental creation of the genetic material of the first living organisms (single-cell prokaryotes) which formed the basis for the evolution to the eukaryotes and more complex organisms (plants and animals).

Constituent genes have the following characteristics:

- They constitute segments of terminal symbols (executable code) and not parameterized functions.
- They have been already evaluated with respect to their usefulness at solving smaller parts of the problem in question during the genes pool creation process.
- They are not divided during the crossover operation.
- They are not partially affected by the mutation operator.
- They are of variable length with predefined minimum and maximum length.

The aim of the constituent genes is to augment the set of the primitive terminal symbols of the grammar with more complex terminals. In the artificial ant problem for example, they enrich the set of the primitive ant's actions (*turn-left*, *turn-right*, and *move*) with more complex behaviours. Where exactly the phenotypes of the constituent genes are added as productions is usually a design decision before a CGE run, especially in cases where that is not obvious. In any situation, the following rules must be satisfied regarding their addition in the grammar:

1. Constituent genes are added as productions in a non-terminal that expands only to terminal symbols or at least to one terminal symbol.
2. The selected non-terminal expands to terminals which are semantically equivalent with the gene's phenotype in order to not violate the correctness of the syntax of the programs which are created with the modified grammar.
3. The number of the non-terminal symbols that expand to more than one production (codon read required) is not increased in order to not make the modified grammar more susceptible than the original grammar to disruption when the genotype changes.

An example of how the phenotypes of the constituent genes are added in the grammar definition is shown in Listing 7.1 and Listing 7.2. Listing 7.1 shows two sample phenotypes of two constituent genes for the artificial ant problem and Listing 7.2 shows the updated version of the <*op*> non-terminal symbol of the BNF-Koza grammar definition (Listing 5.2) after the addition of these phenotypes as production rules.

```
Constituent gene A: ifelse food-ahead [move] [turn-right]   (1)
Constituent gene B: move move turn-left                     (2)
```

Listing 7.1: Sample phenotypes of two constituent genes for the artificial ant problem.

```
<op> ::= turn-left |                                          (1)
         turn-right |                                         (2)
         move |                                               (3)
         ifelse food-ahead [move] [turn-right] |              (4)
         move move turn-left                                  (5)
```

Listing 7.2: The updated *<op>* non-terminal symbol of BNF-Koza after the addition of the phenotypes of the constituent genes of Listing 7.1.

As can be shown in Listing 7.2, the phenotypes of the constituent genes of the Listing 7.1 can be neither divided by crossovers or partially affected by mutations because they constitute undividable phenotype segments expressed as terminal symbols in the production rules (4) and (5) of the *<op>* non-terminal symbol. Section 7.8.1 provides a real sample of a genes pool (Listing 7.25) and the modified grammar definition after the addition of the phenotypes of the constituent genes (Listing 7.26).

In the case of the BNF-Koza grammar it is obvious where constituent genes should be added. A different case is the grammar of Listing 3.7 for the symbolic regression problem. Here there are three candidate non-terminals expanding only to terminal symbols (*<op>*, *<pre-op>* and *<var>*) but only *<var>* is suitable because the constituent genes terminals (valid arithmetic expressions) are semantically equivalent with the productions of only this non-terminal (primitive valid arithmetic expressions). Another similar example is the 3 Multiplexer grammar used in O'Neill and Brabazon (2006a). In this grammar, the constituent genes should be added as productions of the non-terminal *<input>*.

Constituent genes act as an emergent search bias mechanism – the possible hypotheses that can be constructed are not restricted – because they are created randomly based on the original grammar and they are added in the grammar as additional blocks of terminal symbols alongside with the original terminal symbols without removing or replacing productions. The probability that any of the constituent genes will be selected during the genotype-to-phenotype process depends on their number and the number of the productions of the non-terminal in which they are added. For example, in the experiments conducted in this chapter, the pool size parameter is set to 3 and the *<op>* non-terminal symbol of the grammars of the benchmark problems, where the constituent genes are added, have originally already three terminal symbols: *move*, *turn-left*, and *turn-right*.

Therefore, any constituent gene has the same probability that it will be selected with any of the single primitive actions of the original grammar.

Whigham (1995a; 1995b; 1996) was the first to modify a context free grammar as the evolution proceeds by discovering and adding new productions as an example of learnt bias. The grammar is modified in two ways: replacing non-terminals with other non-terminals and terminals in existing productions and adding them as new productions in a non-terminal symbol (without modifying the original productions) or creating new productions that represent underlying structure. In his work, the context-free grammar defines the structure of the initial language in a grammar-based GP system which does not apply a genotype-to-phenotype mapping process. Individuals are represented as derivation trees from the grammar and in order to introduce the new grammar into the population, he uses an operator called *replacement* where a portion of the population is recreated in every generation using the modified grammar.

Another relevant body of work in the area of Genetic Programming, but not in Grammatical Evolution, is that conducted by Ryan, Keijzer and Cattolico (2004) with Run Transferable Libraries (RTLs) and McKay, Hoang, Essam and Nguyen (2006) with Developmental Evaluation (DEVTAG). RTLs are libraries of modules discovered and updated over a number of independent runs transferring knowledge acquired in the past to future evolutionary runs. The main differences with the genes pool used by CGE is that the RTLs contain parameterised functions instead of sets of terminals and that they are trained on "simple" problem instances, in order to tackle more difficult problems later on. Instead, genes pools are recreated in each evolutionary run in CGE. DEVTAG evaluates and compares individuals progressively from simpler instances of the same problem to more difficult instances during their development using tree-adjunct grammar in order to represent the individuals. Instead, CGE applies a random sampling of the problem space to evaluate candidate constituent genes in random parts of the problem without considering the developmental process of the individual.

A similar approach to the concept of constituent genes and their addition in the grammar can be found in Swafford, O'Neill and Nicolau (2011) and Swafford, et al. (2011) where the building blocks are called *modules*. These *modules* are segments of potential useful phenotype and they are added in the BNF grammar during a Grammatical Evolution run as new productions of the *start symbol* (*S*) of the grammar either directly or indirectly

through a special non-terminal called *mod_lib* which acts as a library of *modules*. In CGE, the constituent genes are added always directly as productions in a non-terminal symbol that is expanded to terminals semantically equivalent with the added constituent genes and their probability of being selected is directly related with the semantically equivalent primitive terminals. Instead, the *mod_lib* is a more general approach for adding productions without requiring design decision. Also, the probability of a *module* to be selected depends on the number of productions of the *start symbol* (*S*) of the grammar.

7.2.2 The Behaviour-Switching Concept

The second unique feature of Constituent Grammatical Evolution is the incorporation of the notion of behaviour-switching utilising general problem domain knowledge applicable to agent-oriented systems. Namely, this is the enforcement of a switching of behaviours, through the use of a special BNF grammar definition, which is applicable to agent problems like the Santa Fe Trail.

The motivation behind the enforcement of a behaviour-switching based approach was the intuition that the kind of problems involving agents are not solved efficiently enough with classical evolutionary algorithms because in their very nature these problems are about finding agents which should show a competent behaviour that is comprised of repetitive sequences of actions with re-occurring patterns and not just a meaningless combination of operations. For this reason, a different approach based on evolution of behaviour-switching between *constituents* made up of basic actions should be more suitable for these kinds of problems.

Specifically for the Santa Fe Trail, a series of experiments was conducted (see section 6.5) in order to investigate whether a conditional behaviour-switching approach, implemented in the BNF grammar definition (see Listing 7.3, Listing 6.13, Listing 6.14, and Listing 6.15), could result in an improvement of the effectiveness and/or efficiency of the standard GE algorithm. The experimental results confirmed the correctness of the initial intuition and show a significant improvement on both effectiveness and efficiency. For example, using a conditional behaviour-switching approach – that is the BNF grammar definition shown in Listing 7.3 which defines a search space semantically equivalent with that of the original problem but enforces a search bias toward conditional statements – the success rate (percentage of experiments found a solution) of Grammatical Evolution

increased to 35% with the best solution found requiring only 395 steps (see section 6.5.2 for detailed results of this grammar and its variations), against 10% and 415 steps respectively as shown with the GE using BNF-Koza experiments of section 5.5.

```
N = {behaviour, op, complex-op, single-op, line}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = behaviour

P:
<behaviour>  ::= ifelse food-ahead [ <op> ][ <op> <behaviour> ] |  (1)
                 ifelse food-ahead [ <op> ][ <op> ] |              (2)
                 <op>                                              (3)
<op>         ::= <single-op> |
                 <single-op> |
                 <single-op> |
                 <complex-op> |
                 <complex-op> |
                 <complex-op>
<complex-op> ::= <line> |
                 <complex-op> <line>
<line>       ::= ifelse food-ahead [ <complex-op> ][ <complex-op> ] |
                 <single-op>
<single-op>  ::= turn-left |
                 turn-right |
                 move
```

Listing 7.3: Example of a fixed behaviour-switching BNF grammar definition for the artificial ant problem (see also section 6.5 for variations of this grammar).

With this approach, the resultant ant control programs are enforced, via the behaviour-switching BNF grammar definition, to perform a check (condition) whether there is food ahead and if yes to perform the appropriate *constituent behaviour*. When no food is found ahead, the control program is forced to take some other alternative *constituent behaviour* and/or perform a new check, and so on.

As *constituent behaviour*, any single or complex action of the agent is defined in this context. In the BNF grammar definition of Listing 7.3, it is represented with the non-terminal symbol <*op*>. As can be seen, the <*op*> non-terminal in this grammar can be translated either directly to a single action (<*single-op*>) or to a set of multiple actions (<*complex-op*>) with the same probability (50%).

The production rules (1) and (2), of the non-terminal symbol <*behaviour*>, are the important points where the behaviour-switching approach is imposed in this grammar definition. Namely, when <*behaviour*> is translated to one of these production rules, the

agent is enforced to perform the previously mentioned check (condition) of whether there is food ahead or not. Depending on the selected production rule, the agent will be imposed to proceed with subsequent checks (production rule number one) or not (production rule number two). In this way, a kind of memory is incorporated indirectly in the ant's behaviour, because the resultant control program can be expressed as a simple finite-state machine of the form shown in Listing 7.4.

```
If food ahead                          (State 1)
  execute constituent behaviour 1
Else
  execute constituent behaviour 2
  If food ahead                        (State 2)
    execute constituent behaviour 3
  Else
    execute constituent behaviour 4
    If food ahead                      (State 3)
      execute constituent behaviour 5
    Else
      ..........
      ..........                       (State N)
    End If
  End If
End If
```

Listing 7.4: Behaviour-switching represented as pseudo-code with a corresponding FSM.

The grammar definition example of Listing 7.3 for the artificial ant problem defines a search space which is semantically equivalent with the search space of the original problem as defined by Koza (1992) due to the fact that this grammar is the BNF-Koza grammar definition of Listing 5.2 wrapped by the conditions of the production rules (1) and (2) of the <*behaviour*> non-terminal symbol in order for a declarative search bias to be stated. The production rule (3) of the <*behaviour*> non-terminal symbol exists to allow the bypassing of the condition checking enforcement.

The behaviour-switching BNF grammar definition used by CGE in the benchmark problems of this work is of the form of the BNF grammar definition shown in Listing 7.3 with the following modifications in the productions of the <*op*> non-terminal symbol: the three <*single-op*> productions are replaced by the three terminals *turn-left*, *turn-right*, *move* and the <*complex-op*> non-terminal symbols with the phenotypes of the constituent genes. If the constituent genes are more than three, they are added as additional productions of the <*op*> non-terminal. This modified grammar states a declarative language bias which can be expanded to areas of the excluded search space – where good

solutions may exist – with the addition of the constituent genes before a Grammatical Evolution run.

Whigham (1996) was the first to investigate the impact and the potential benefits of grammatical bias in a grammar-based GP system where a context-free grammar is used to define the structure of the individuals. He demonstrates the use of declarative bias to modify the search space by adjusting the grammar so that it represents more closely the believed solution and he shows the importance of this as it gives a clear statement of bias which is external to the learning system.

7.2.3 Genotype Bloating Elimination

A third feature which differentiates Constituent Grammatical Evolution from the standard Grammatical Evolution algorithm is the incorporation of a limit to the genotype size of the individuals. After each crossover operation, the length of the offspring is checked against this limit. If the limit is exceeded, the genotype of the offspring is pruned, starting from the end, until it is reduced to the maximum allowed size.

With the enforcement of this limit, the genotype bloating phenomenon is tackled even though GE literature mentions that this phenomenon could be beneficial because it reduces the effect of the destructive crossovers (O'Neill, Ryan, Keijzer and Cattolico, 2003). In Constituent Grammatical Evolution, as discussed in section 7.2.1, the impact of destructive crossover events is reduced through the utilisation of modularity / building blocks (Angeline and Pollack, 1993; Rosca and Ballard, 1994; Hemberg, 2010).

The elimination of the genotype bloat increases the performance of the evolutionary algorithm in terms of required processing power because the decrease of the size of the genotype of the individuals leads to fewer non-terminal expansion-to-production operations during the genotype-to-phenotype mapping process of an individual and consequently to less average mapping time. This is demonstrated in section 7.7.4 and further investigated and discussed in section 7.8.3 where this feature is benchmarked using different limits on the Santa Fe Trail with Grammatical Evolution using BNF-Koza.

7.3 CGE Algorithm Description

The Constituent Grammatical Evolution algorithm uses four inputs as depicted in Figure 7.1: a problem specification, a language specification, a behaviour-switching specification, and the grammatical evolution algorithm.

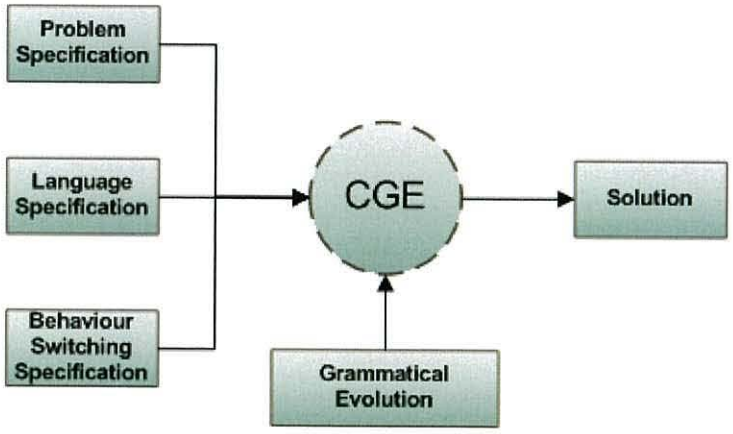


Figure 7.1: Constituent Grammatical Evolution inputs.

- Problem Specification (*PS*) is the program which simulates the problem in question and assigns to each individual and constituent gene a fitness value.
- Language Specification (*BNF-LS*) is the original unbiased BNF grammar definition of the problem in question which dictates the grammar of the output programs of the individuals which are executed by the problem specification simulator. It is used for the genotype-to-phenotype mapping of the constituent genes.
- Behaviour-Switching Specification (*BNF-BS*) is the BNF grammar definition which modifies the Language Specification (*BNF-LS*) by incorporating in the original specification (*BNF-LS*) the switching behaviour approach – one of the unique features of CGE. Namely, the *BNF-BS* specification is a BNF grammar definition which enforces a conditional check before each agent's action and states a declarative language bias in order to restrict the search space to potentially more useful areas.
- Grammatical Evolution Algorithm (*GE*) is the evolutionary algorithm which is used for the evolution of the population of the individuals.

In addition with these inputs, Constituent Grammatical Evolutions is configured with the parameters described in Listing 7.5.

- Individual Max Codons (*IMC*):
  The maximum allowed number of codons of an individual during a GE evolutionary run. If the value of *IMC* is zero (0), then no size limit is enforced on the genotype.
- Genes Pool Size (*S*):
  The size of the pool of the constituent genes (namely, the number of constituent genes used for the construction of the phenotype of an individual). This size must be less than the number of all possible decimal values of a codon's bit string, e.g. if the codon size is 8, then the pool size *S* must be less or equal to 256 - *y*, where *y* is the number of the basic/elementary terminal symbols of the *BNF-LS* grammar.
- Gene Generations (*G*):
  How many times populations of genes are randomly created. Dictates the total number (*M*) of randomly created genes. This number (all candidate genes created) is $M = S * G$.
- Gene Evaluation Iterations (*L*):
  The iterations of the execution of the program of a candidate constituent gene in order to determine the interim (temporary) fitness value of this gene.
- Gene Evaluations (*E*):
  The number of times a candidate constituent gene is evaluated (interim fitness values).
- Gene Codons Min (*CMin*):
  The minimum allowed size in codons of a randomly created candidate constituent gene.
- Gene Codons Max (*CMax*):
  The maximum allowed size in codons of a randomly created candidate constituent gene.
- Gene Max Wraps (*W*):
  The maximum number of wraps allowed during the genotype-to-phenotype mapping of a constituent gene.

Listing 7.5: CGE configuration parameters.

The overall description of the Constituent Grammatical Evolution algorithm is depicted in Listing 7.6. Namely, using the previously described inputs and the CGE specific parameters (*IMC*, *S*, *G*, *L*, *E*, *CMin*, *CMax*, and *W*) as defined in Listing 7.5, a pool *P* of constituent genes is created and filled in the following way. *S* genes are randomly created which are represented as binary strings. These genes will be candidates for the pool of the constituent genes. The minimum length of each string is *CMin* codons and the maximum length is *CMax* codons (the size in bits of each codon is specified by the Grammatical Evolution algorithm; usually its value in the Grammatical Evolution literature is eight).

```
// Creation of the constituent genes pool
Set current constituent genes pool P of size S
Repeat G times
  Create a new empty pool P'
  Repeat until P' is full
    Create gene N with genotype size C codons (CMin ≤ C ≤ CMax)
    Map genotype of N to phenotype using BNF-LS and W max wraps
    Set i = 0
    While i < E
      Put N in a random environment location of PS
      Set direction of N randomly
      Repeat L times
        Execute phenotype (program) of N
      End Repeat
      Calculate and set interim fitness value Vi of gene N
      Set i = i + 1
    End While
    Calculate and set final fitness value F of gene N
    Add N to P'
  End Repeat
  Create a pool T of size 2*S
  Add to T the genes of P and P'
  Sort T according the fitness values F of genes
  Create an empty P''
  Add to P'' the best S genes of T
  Replace P with P''
End Repeat

// Addition of the constituent genes in the original grammar
For each gene N in P
  Add phenotype of N in BNF-BS grammar definition as a_
    production to a non-terminal which expands to semantically_
    equivalent terminal symbols.
End For

// Grammatical Evolution run
Create population R of individuals
While solution not found and max generations limit not exceeded
  Evolve population R with Grammatical Evolution using BNF-BS_
    grammar and enforcing IMC Limit to individuals' genotypes
End While
Retrieve solution K
```

Listing 7.6: The Constituent Grammatical Evolution algorithm.

The fitness function for the evaluation of the candidate constituent genes depends on the problem specification and is similar to the fitness function used for the individuals in the problem in question. The general concept behind the creation and evaluation of the constituent genes is to find reusable modules from which the genotype of the individual will be constructed. For demonstration purposes, the way constituent genes are evaluated (described next) is based on the implementation in the Santa Fe Trail problem.

The fitness value F of each gene is calculated as follows. First, the gene's genotype is mapped to its phenotype using the language specification *BNF-LS* (BNF grammar

definition) and the Grammatical Evolution mapping formula with W genotype wraps limit. Then, the gene is placed in a random location of the environment with a random heading. The gene's code (the phenotype as for an individual) is executed for L times and evaluated to produce an interim fitness value. In this way, the control program runs for a while before the fitness value is calculated (in case of the Santa Fe Trail problem, number of pieces of food found during the run). Therefore, the gene has the chance to be tested in a larger part of the trail if randomly placed on or near the trail, and consequently the fitness value is much more indicative about the effectiveness and efficiency of the control program (gene) than if it were allowed to run its code (control program) only once.

Because the genes are placed randomly in the environment before each interim evaluation, in order to get a representative fitness value of the gene, the above interim evaluation (calculation of the interim fitness value) takes place for a number of times. Consequently, the interim fitness value is calculated E times. Then, the final fitness value of the candidate constituent gene is calculated with the formula shown in Equation 7.1

Equation 7.1: Final fitness value calculation

$$F = ((V_1 + \ldots + V_N)/N) \times (V_{max}/S)$$

where: F is the final fitness value of the constituent gene used for comparison with other genes and for deciding whether it will placed in the genes pool or not; N ($N \leq E$) is the number or interim evaluations with fitness value > 0; V_n is the Interim fitness value of the interim evaluation n ($0 \leq n \leq N$); V_{max} is the maximum interim fitness value found; and S is the number of steps performed during this interim evaluation.

The reason for evaluating the gene for a number of times E is that it is placed randomly in the environment and consequently there is the chance of a potential "good" gene being assigned a low fitness value; for example in case of the Santa Fe Trail problem, a "good" gene placed away from the trail probably will be assigned with a very poor interim fitness value. The average interim value $(V_1 + \ldots + V_N)/N$ is calculated by taking into account only the evaluations (interim fitness values) with value greater than 0. In this way, all cases where the gene was randomly placed away from useful areas, and for this reason had no chance to get a positive fitness value, are ignored. Namely, in case of the Santa Fe Trail problem, if the gene was placed away from the trail and had no chance of finding food, then the gene would be actually biased negatively due to its zero fitness value even

though the gene could be useful and able to find pieces of food if it were placed near or on the trail. Consequently, by taking into account the calculation of the average fitness value only evaluations greater than zero, this is much more indicative of the general performance of the gene. Indeed, the addition in the formula of the (V_{max} / S) calculation, promotes genes which are more efficient (for example, found more food with fewer steps). For this calculation, the highest interim fitness value of the gene found by all interim evaluations and the corresponding steps required by the gene during this evaluation are taken into account.

The genes of each generation are compared with the genes of the previous generation according to their fitness value and the best S genes of these two generations are placed in the pool P replacing the existing genes. This will be repeated for G generations. Finally, the pool P will be filled with the best S genes (and their corresponding phenotypes) created during the G generations.

When the genes pool is finally created and filled, the phenotypes of the constituent genes are added as terminal operators in the behaviour-switching BNF grammar definition (*BNF-BS*). Then an initial population of individuals is created randomly where each individual consists of a "control gene" which dictates which original terminal operators and which added terminal operators (phenotypes of the constituent genes) of the *BNF-BS* grammar definition are used and in what order.

Finally, the population of individuals is evolved – until a solution is found or a maximum limit of generations is reached – using the standard Grammatical Evolution algorithm with the *BNF-BS* grammar definition (augmented by the constituent genes phenotypes) and enforcing the individual's genotype size maximum limit *IMC* after each crossover.

7.4 Application of CGE to the Artificial Ant Problem

As noted, the Constituent Grammatical Evolution algorithm, unlike Grammatical Evolution, uses two distinct BNF grammar definitions: the first, for the genotype-to-phenotype mapping of the candidate constituent genes; and the second for the individuals. The BNF-Koza grammar definition (see Listing 5.2), which is used by Grammatical Evolution in the Santa Fe Trail problem for the genotype-to-phenotype mapping of the artificial ants, is used by Constituent Grammatical Evolution for the genotype-to-

phenotype mapping of the constituent genes. Instead, for the mapping of the artificial ants, the conditional behaviour-switching BNF grammar definition (*BNF-BS*) is used, which results from a template (*BNF-BS* Blueprint) with the addition of the phenotypes of the constituent genes.

The tuples {*N*, *T*, *P*, *S*} of the BNF-Koza, *BNF-BS* Blueprint, and *BNF-BS* grammar definitions are depicted in Listing 5.2, Listing 7.7, and Listing 7.8 respectively, where *N* is the set of non-terminal symbols, *T* the set of terminal symbols, *P* a set of production rules that maps the elements of *N* to *T*, and *S* is a start symbol that is a member of *N*.

```
N = {behaviour, op}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = behaviour
P:
<behaviour> ::= ifelse food-ahead [ <op> ][ <op> <behaviour> ] |
                ifelse food-ahead [ <op> ][ <op> ] |
                <op>
<op> ::= turn-left |
         turn-right |
         move
```

Listing 7.7: BNF-BS Blueprint grammar definition for the artificial ant problem.

```
N = {behaviour, op}
T = {turn-left, turn-right, move, ifelse, food-ahead, [, ]}
S = behaviour
P:
<behaviour> ::= ifelse food-ahead [ <op> ][ <op> <behaviour> ] |
                ifelse food-ahead [ <op> ][ <op> ] |
                <op>
<op> ::= turn-left |
         turn-right |
         move |
         ... {constituent genes phenotypes}(*)
```

Listing 7.8: BNF-BS grammar definition for the artificial ant problem. The phenotype of every constituent gene (*) is added as a production rule in the <op> non-terminal symbol.

The *BNF-BS* grammar definition (Listing 7.8) is constructed dynamically at the beginning of an evolutionary run, from a template (*BNF-BS* Blueprint of Listing 7.7) where the phenotypes of the selected randomly created constituent genes are added as production rules in the <*op*> non-terminal symbol, in the same way as in the examples of Listing 7.1 and Listing 7.2.

The search space defined by this dynamically created *BNF-BS* grammar definition is always a subset of the original search space which is defined, as shown, by the BNF-Koza grammar (Listing 5.2). This subset depends on the added phenotypes of the constituent genes and differs from these of other dynamically created *BNF-BS* grammars if the added constituent genes are different as well. Because the mapping process of the constituent genes uses the BNF-Koza grammar definition, every possible program created by BNF-Koza can be created with some dynamically created instance of the *BNF-BS* grammar and vice versa. Consequently, the search space defined by all possible *BNF-BS* grammar instances is semantically equivalent with that of the original problem (BNF-Koza) but declaring a different search bias. Instead, each instance of the *BNF-BS* grammar states additionally a specific language bias which depends on the added phenotypes of the constituent genes.

Figure 7.2 depicts the overall architecture of the implementation of Constituent Grammatical Evolution in the artificial ant problem and how it relates with the standard Grammatical Evolution algorithm.

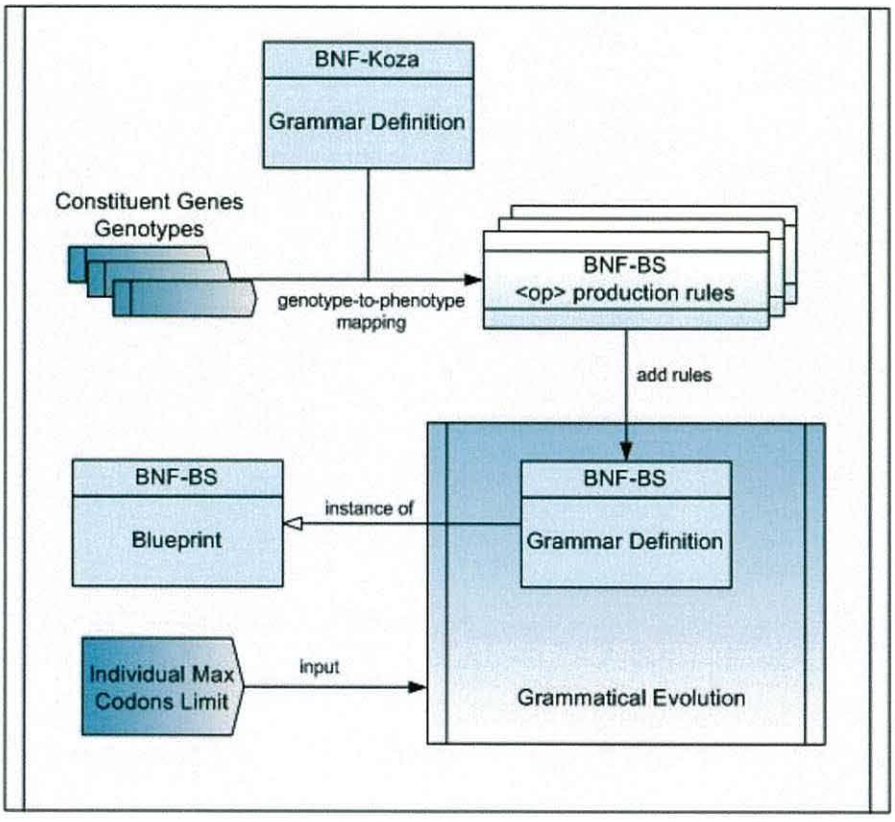


Figure 7.2: The Constituent Grammatical Evolution (CGE) system.

Figure 7.3 shows the BNF grammar definitions used by the Constituent Grammatical Evolution and the Grammatical Evolution algorithm (both GE using BNF-Koza and GE using BNF-O'Neill) as well as their relations and how they are used in each case.

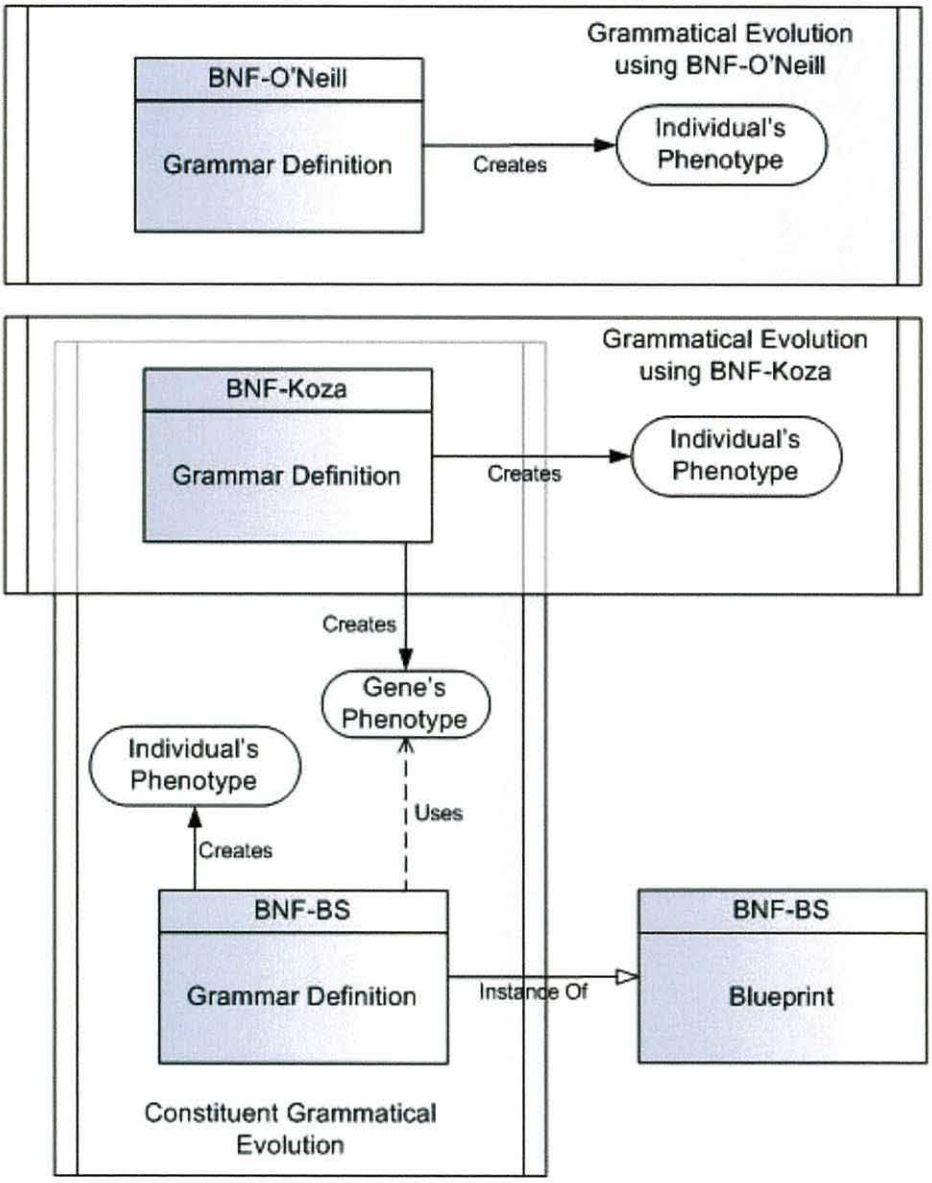


Figure 7.3: BNF grammar definitions used by GE and CGE systems.

Finally, for the execution of the CGE benchmarking experiments described in the next sections, the NetLogo model used in the evolutionary experiments of Chapter 5 and presented in section 5.3.2, has been extended. Namely, the Constituent Grammatical Evolution algorithm has been implemented with the NetLogo programming language and controls have been added to the interface of the model for setting the parameters of the

Constituent Grammatical Evolution algorithm. A screenshot of this model for the Santa Fe Trail problem can be shown in Figure 7.4.

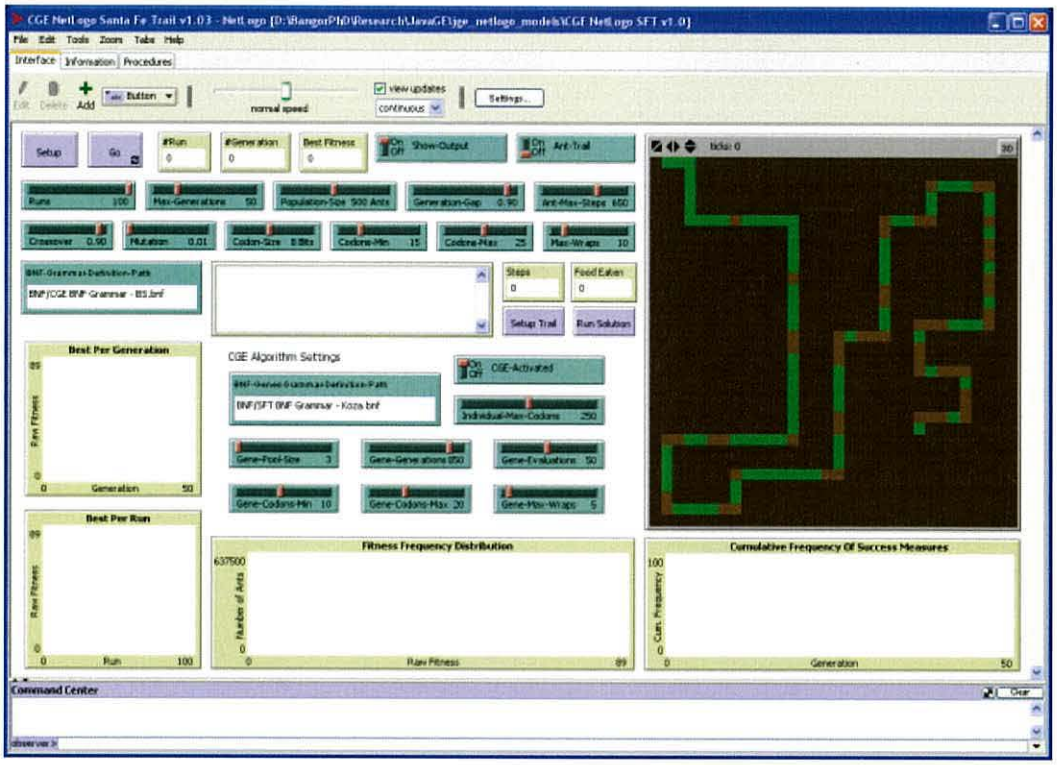


Figure 7.4: Interface of the Evolutionary SFT model for CGE and GE.

This extended model provides the option of using Grammatical Evolution or Constituent Grammatical Evolution for the evolution of the population of ants in the Santa Fe Trail problem, by switching "Off" or "On" respectively the "CGE-Activated" switch. The BNF grammar definition for the genotype-to-phenotype mapping of the constituent genes is defined in the "BNF-Genes-Grammar-Definition-Path" text box. Using the interface of the model, the modeller can also set the CGE parameters with the "Individual-Max-Codons", "Gene-Pool-Size", "Gene-Generations", "Gene-Evaluations", "Gene-Codons-Min", "Gene-Codons-Max", and "Gene-Max-Wraps" sliders.

7.5 Benchmarking CGE on the Santa Fe Trail Problem

7.5.1 Experiments Setup

The performance of the Constituent Grammatical Evolution algorithm has been compared against Grammatical Evolution in the Santa Fe Trail problem in order to discover whether

this new algorithm improves the later in a standard Genetic Programming benchmark. Because of the Grammatical Evolution literature issue of benchmarking against GP with the use of a grammar which states a declarative language bias and excludes good solutions (BNF-O'Neill), as demonstrated and discussed in Chapter 5, Constituent Grammatical Evolution effectiveness and efficiency have been benchmarked against both GE using BNF-Koza and GE using BNF-O'Neill.

For these benchmarks, three series of experiments were used to evaluate CGE, GE using BNF-Koza, and GE using BNF-O'Neill. In each series of experiments, five distinct experiments were conducted consisting of one hundred evolutionary runs each. Namely, a total of five hundred evolutionary runs were performed for each algorithm. Note here that the GE using BNF-Koza experiments and the GE using BNF-O'Neill experiments are these described in the section 5.4 of the previous chapter. The tableau in Table 7.1 shows the Grammatical Evolution settings and parameters of each evolutionary run and the Table 7.2 shows the CGE specific parameters.

Table 7.1: Grammatical Evolution tableau for the Santa Fe Trail.

Objective	Find a computer program in the NetLogo programming language to control an artificial ant so that it can find all 89 pieces of food located on the Santa Fe Trail.
Terminal Operators	*turn-left, turn-right, move, food-ahead*, plus *constituent genes phenotypes* when the CGE algorithm is used instead of the standard GE algorithm.
Raw Fitness	Number of pieces of food picked up before the ant times out with 650 operations.
BNF Grammar	CGE: BNF-Koza (original) for Genes and BNF-BS (behaviour-switching) for Ants. GE using BNF-Koza: BNF-Koza. GE using BNF-O'Neill: BNF-O'Neill.
Evolutionary Algorithm	Steady-State Genetic Algorithm, Generation Gap = 0.9, Selection Mechanism: Roulette-Wheel Selection.
Initial Population	Randomly created with the following restrictions: Minimum Codons = 15 and Maximum Codons = 25.
Parameters	Population Size = 500, Maximum Generations = 50, Mutation Prob. = 0.01, Crossover Prob. = 0.9, Codon Size = 8, Wraps Limit = 10.

Table 7.2: CGE settings for the Santa Fe Trail.

Parameter	Value	Parameter	Value
Codons Limit, *IMC*	250	Gene Evaluations, *E*	50

Gene Pool Size, S	3	Gene Codons Min, *CMin*	10
Gene Generations, G	850	Gene Codons Max, *CMax*	20
Gene Code Iterations, L	10	Gene Max Wraps, W	5

The BNF grammars used in these experiments are BNF-Koza (Listing 5.2), BNF-O'Neill (Listing 5.3), and BNF-BS (Listing 7.8). The fitness value of the candidate constituent genes is calculated using the formula in Equation 7.1.

Note that because CGE uses the standard Grammatical Evolution algorithm for the evolution of the individuals, most GE settings used in all experiments are the same with these used to benchmark GE using BNF-Koza in Chapter 5 (see Table 5.2) except: a) the BNF grammar definition, because CGE uses two BNF grammar definitions as described in the previous section and because the standard Grammatical Evolution algorithm is evaluated using two different BNF Grammars (BNF-Koza and BNF-O'Neill); and b) the terminal operators, because CGE introduces the concept of constituent genes which are expressed as compositions of the initial terminal operators, which are added to the terminal operators of the original problem specification (grammar definition).

Regarding the eight new parameters introduced by CGE, the Gene Pool Size S and Codons Limit *IMC* affect directly the Grammatical Evolution algorithm because they modify the grammar used by GE and restrict the genotype size of the individuals created during the GE run respectively, as described in detail in section 7.3. The other six parameters concern the creation of the genes pool and can be thought as an independent mechanism for the initial identification of useful building blocks in order to modify the grammar before a GE run starts.

The values of the CGE parameters have been decided as follows: For Gene Pool Size S, the experimental results of the section 6.4, show that using three building blocks is a reasonable choice which can potentially result in an improvement in success rate. Additionally, these experiments show that the use of more than three building blocks does not result in a proportional performance improvement in terms of success rate and solution quality (future investigation and experimentation is required to explore in more detail). Instead, more computational effort would be required for the creation and evaluation of the candidate constituent genes before each run in order to increase the chance that all of the genes of the pool be useful building blocks. The Codons Limit *IMC* has been set to

250 codons in order to eliminate the genotype bloat phenomenon without risking an impact on the success rate. The experiments of GE using fixed-length genomes in Chapter 5 allow the assumption that restricting the maximum size to 250 codons can be considered as a safe limit. Further work is required for this parameter as well to find more suitable values.

The values of Gene Codons Min *CMin*, Gene Codons Max *CMax* and Gene Max Wraps *W*, specify the size of the genomes of the candidate constitute genes and the number of the allowed wraps during their genotype-to-phenotype mapping process using the original grammar. These are decided in respect to the values of the corresponding parameters of the GE run setup, assuming that slightly smaller values would be appropriate taking into account that constituent genes aim to produce smaller phenotypes than the individuals (partial solutions). It is also noted that the codon size of the constituent genes is defined by the GE setup (and the same with the individuals), and has the value eight in the experiments of this work. The last three parameters Gene Generations *G*, Gene Evaluations *E*, and Gene Code Iterations *L* are decided using arbitrary large values in order to increase the chance useful constituent genes (building blocks) to be found for the genes pool.

Finally, it is noted that besides the mentioned criteria for the selection of the values of the CGE parameters, no further attempt was made to optimise them because the aim of this work is to investigate whether this algorithm has the potential for improving the Grammatical Evolution algorithm. Promising results indicate the usefulness of this approach and further investigation and optimisation of parameters is warranted.

7.5.2 Experiments Results

Constituent Grammatical Evolution was successful at finding a solution in the Santa Fe Trail problem with very high success rates, ranging from 85% to 94% with an average success rate of 90%. The best solution found by CGE, requires only 337 steps and there is no Genetic Programming or Grammatical Evolution publication in the knowledge of the author presenting a solution requiring less steps. The NetLogo code for this solution is shown in Listing 7.9.

Table 7.3: CGE experimental results in the Santa Fe Trail problem.

	Exp #1	Exp #2	Exp #3	Exp #4	Exp #5	**Best**
Runs	100	100	100	100	100	**100**
Steps	393	375	393	377	337	**337**
Success	85%	93%	89%	94%	87%	**94%**
Avg. Success	**90%**					

The Table 7.3 shows the detailed results of the CGE experiments. The column "Best" shows the best value of all five experiments. "Runs" is the number of evolutionary runs performed in the experiment, "Steps", the required steps of the best solution found in the particular experiment, "Success", how many evolutionary runs (percentage) found a solution, and "Avg. Success", the average success rate of all five experiments.

```
ifelse food-ahead
  [move]
  [turn-right
   ifelse food-ahead
     [ifelse food-ahead
       [ifelse food-ahead
         [move move]
         [move]
        ifelse food-ahead
          [move move]
          [move]
       ]
       [turn-left]
      move
     ]
     [ifelse food-ahead
        [move]
        [turn-right]
      ifelse food-ahead
        [turn-left]
        [turn-right
         ifelse food-ahead
           [move]
           [ifelse food-ahead
             [move]
             [turn-right]
            move
           ]
        ]
     ]
  ]
```

Listing 7.9: NetLogo code of the best solution found by CGE in the Santa Fe Trail problem. This solution requires 337 steps.

The results of the experiments using the standard GE with BNF-Koza and BNF-O'Neill can be seen in Table 7.4 and in Table 7.5. These are the experimental results presented in section 5.5 and are reprinted here for the convenience of the reader. A cumulative frequency measure of success over 500 runs of CGE and GE can be seen in Figure 7.5.

Table 7.4: GE using BNF-Koza experimental results in Santa Fe Trail.

	Exp #1	Exp #2	Exp #3	Exp #4	Exp #5	**Best**
Runs	100	100	100	100	100	**100**
Steps	419	507	415	541	479	**415**
Success	8%	11%	10%	6%	13%	**13%**
Avg. Success	**10%**					

Table 7.5: GE using BNF-O'Neill experimental results in Santa Fe Trail.

	Exp #6	Exp #7	Exp #8	Exp #9	Exp #10	**Best**
Runs	100	100	100	100	100	**100**
Steps	609	609	607	609	607	**607**
Success	80%	76%	75%	81%	74%	**81%**
Avg. Success	**78%**					

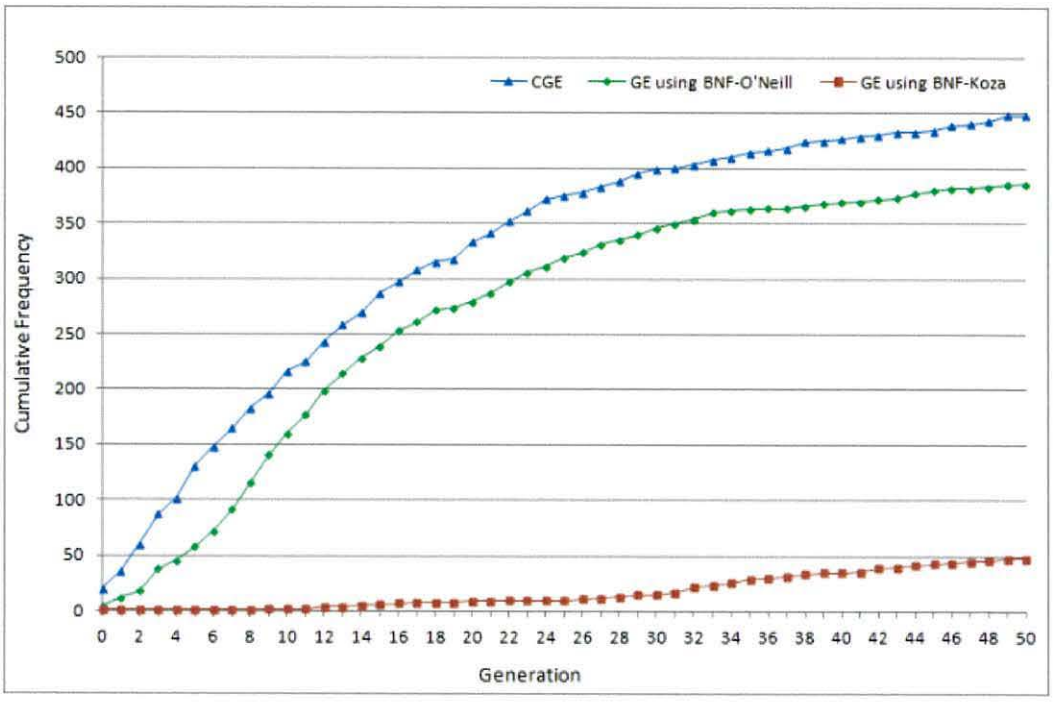


Figure 7.5: CGE vs. GE on the Santa Fe Trail problem. CGE has already created the genes pool and modified the grammar before generation zero.

The experimental results show that Constituent Grammatical Evolution improves Grammatical Evolution in terms of success rate whether the later uses a BNF grammar definition (BNF-Koza) which defines a search space semantically equivalent with that used in the original problem (Koza, 1992), or whether it uses a BNF grammar definition (BNF-O'Neill) which defines a restricted search space. Additionally, Constituent Grammatical Evolution is able to find much better solutions in terms of the required steps. It is noted here in respect to the performance of CGE displayed in the diagram of Figure 7.5 that CGE creates and evaluates constituent genes before each evolutionary run.

Additionally, the statistics (Table 7.6) of the experimental results of all three series of experiments show that Constituent Grammatical Evolution improves the standard Grammatical Evolution in one more aspect, the genotype size (smaller genomes).

Table 7.6: Statistics of CGE benchmarking experimental results in the Santa Fe Trail problem.

Description	**GE using BNF-Koza**	**GE using BNF-O'Neill**	**CGE**
Success rate of finding a solution	10%	78%	**90%**
Best solution found in terms of ant's steps	415 steps	607 steps	**337 steps**
Average ant's steps of all solutions found	584 steps	615 steps	**448 steps**
Best solution found in terms of genotype size (bits)	142 bits	90 bits	**73 bits**
Average genotype size of solutions found (bits)	1,568 bits	423 bits	**419 bits**
Average genotype size of best individual per run (bits)	2,923 bits	816 bits	**492 bits**
Percentage of solutions with genotype size > 2000 bits	27.08%	2.33%	**0%**
Percentage of best individuals per run with genotype > 2000 bits	58.80%	11.20%	**0%**
Best solution found in terms of phenotype size (operators)	**9 op.**	**9 op.**	13 op.
Average phenotype size of solutions found (operators)	**15 op.**	24 op.	28 op.
Average number of ants programs	24,644 ants	12,130 ants	**9,487ants**

created and validated in a run			
Average duration of an evolutionary run (secs)	347sec	**109 sec**	121 sec
Average duration of a generation creation and evaluation (secs)	7.10 sec	**4.46 sec**	7.01 sec

One aspect where the standard Grammatical Evolution outperforms CGE is the phenotype size. Solutions created by the standard Grammatical Evolution algorithm are of smaller phenotype size than these found by CGE. The smallest solutions found by Grammatical Evolution and CGE consist of 9 and 13 terminal operators respectively.

The observed large difference in the "Average number of ants programs created and validated in a run" between CGE, GE using BNF-Koza, and GE using BNF-O'Neill, is related to the differences in the success rates of these setups. Namely, higher success rates result in runs with fewer generations (the run ends because a solution found) and consequently the average cumulative size of the populations of all generations in a run decreases accordingly. Also, it is noted that the additional execution time required before each GE run by CGE for the creation of the genes pool is taken into account in the displayed duration figures. The computational effort of CGE is analysed in section 7.7.4.

The promising results of CGE benchmarking on the Santa Fe Trail problem raised the question whether it can improve Grammatical Evolution in other problems as well. For this reason, CGE was applied and benchmarked on three additional problems. The first is a more difficult version of the Santa Fe Trail problem and the two other are typical maze searching problems.

7.6 Application to more Problems

7.6.1 The Los Altos Hills Problem

The performance of the Constituent Grammatical Evolution algorithm has been benchmarked against the Grammatical Evolution algorithm again, this time in the Los Altos Hills problem which is described in detail in section 2.5.4. For the performance of the experiments, a modified NetLogo model of this presented in section 7.4 (Figure 7.4) was used. Namely, the Santa Fe Trail has been replaced by the Los Altos Hills trail. The interface of this model is shown in Figure 7.6.

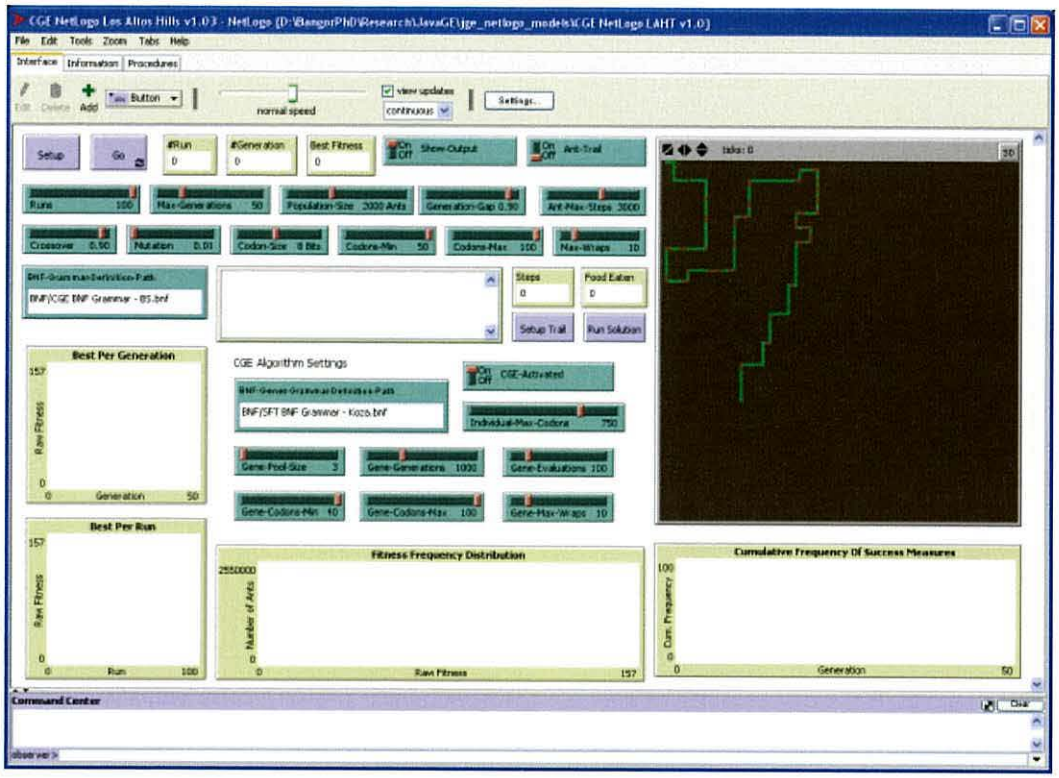


Figure 7.6: Interface of the Evolutionary Los Altos Hills model for CGE and GE.

In order to evaluate CGE against GE, a series of three experiments was conducted (CGE, GE using BNF-Koza, and GE using BNF-O'Neill), consisting of 100 evolutionary runs each using the configuration shown in Table 7.7 and in Table 7.8. The constituent genes evaluation formula used in this problem was the same as that used on the Santa Fe Trail (Equation 7.1).

Table 7.7: Grammatical Evolution Tableau for Los Altos Hills.

Objective	Find a computer program in the NetLogo programming language to control an artificial ant so that it can find all 157 pieces of food located on the Los Altos Hills trail.
Terminal Operators	*turn-left, turn-right, move, food-ahead*, plus *constituent genes phenotypes* when the CGE algorithm is used instead of the standard GE algorithm.
Raw Fitness	Number of pieces of food picked up before the ant times out with 3000 operations.
BNF Grammar	CGE: BNF-Koza (original) for Genes and BNF-BS (behaviour-switching) for Ants. GE using BNF-Koza: BNF-Koza. GE using BNF-O'Neill: BNF-O'Neill.
Evolutionary Algorithm	Steady-State Genetic Algorithm, Generation Gap = 0.9, Selection Mechanism: Roulette-Wheel Selection.
Initial	Randomly created with the following restrictions:

Population	Minimum Codons = 50 and Maximum Codons = 100.
Parameters	Population Size = 2000, Maximum Generations = 50, Mutation Prob. = 0.01, Crossover Prob. = 0.9, Codon Size = 8, Wraps Limit = 10.

Table 7.8: CGE settings for Los Altos Hills.

Parameter	Value	Parameter	Value
Codons Limit, *IMC*	750	Gene Evaluations, *E*	100
Gene Pool Size, *S*	3	Gene Codons Min, *CMin*	40
Gene Generations, *G*	1000	Gene Codons Max, *CMax*	100
Gene Code Iterations, *L*	10	Gene Max Wraps, *W*	10

The CGE parameters concerning the creation and evaluation of the candidate constituent genes (except Gene Code Iterations L) and the genotype size limit have been assigned with larger values in this problem than in the setup of the Santa Fe Trail problem because of the higher difficulty of Los Altos Hill. The other two parameter, Gene Pool Size S and Gene Code Iterations L, are used with the same values as in the Santa Fe Trail.

The BNF-BS grammar definition used in this problem is shown in Listing 7.10. Note that a slightly different BNF-BS grammar definition is used than used for the Santa Fe Trail problem. Namely, in the *<behaviour>* non-terminal symbol a new production rule is added (the first production rule which appears twice) which increases the search space allowing a sequence of *<behaviour>* non-terminals. Indeed, both the first and the second conditional production rules appear twice each in order to bias the search toward a conditional behaviour. These modifications in the BNF-BS grammar definition were performed because the specifications of the Los Altos Hills require individuals with a larger number of time steps performed per piece of food. Namely, the Los Altos Hills problem in order to be solved (according to its specifications) requires less efficient individuals in terms of the fraction of steps and food pieces. The fractions of available time steps and total amount of food are 7.3 (650/89) and 19.2 (3000/156) for the Santa Fe Trail and the Los Altos Hills problems respectively.

```
N = {behaviour, op}
T = {turn-left, turn-right, move, ifelse,
     food-ahead, [, ]}
S = behaviour
P:
<behaviour> ::=
       <behaviour> ifelse food-ahead [<op>][<op> <behaviour>] |
```

```
        <behaviour> ifelse food-ahead [<op>][<op> <behaviour>] |
        ifelse food-ahead [ <op> ][ <op> <behaviour> ] |
        ifelse food-ahead [ <op> ][ <op> <behaviour> ] |
        ifelse food-ahead [ <op> ][ <op> ] |
        <op>
<op> ::= turn-left |
         turn-right |
         move |
         ... {constituent genes phenotypes}
```

Listing 7.10: The BNF-BS grammar definition for Los Altos Hills.

The experimental results are shown in Table 7.9. "Runs" is the number of evolutionary runs performed in the experiment; "Best Solution's Steps", the required steps (ant moves) of the best solution found in terms of efficiency (required steps) in the particular experiment; and "Success Rate", how many evolutionary runs (percentage) found a solution.

Table 7.9: Experimental results of the Los Altos Hills problem.

	CGE	GE BNF-Koza	GE BNF-O'Neill
Runs	100	100	100
Best Solution's Steps	1093	No solution	No solution
Success Rate	**9%**	**0%**	**0%**

Even though the Los Altos Hill is a much more challenging problem than the Santa Fe Trail, CGE managed to find a solution in contrast to GE which was not able to find one.

Figure 7.7 shows a graph which depicts the fitness value of the best individual, in terms of pieces of food found, for each evolutionary run. In this graph it is observed that most of the best individuals found by GE using BNF-O'Neill and some of the best individuals found by GE using BNF-Koza were able to find 116 pieces of food, which means that some of the best individuals found in these cases were probably able to solve only the SFT-like part of the trail (or at least most of it) but could not tackle the new irregularities introduced in the Los Altos Hills trail. Also, in the runs where CGE did not find a solution for the Los Altos Hills trail, the best individuals were able to find at least 116 pieces of food. Namely, these individuals were probably able to solve the SFT-like part of the trail or at least most of it.

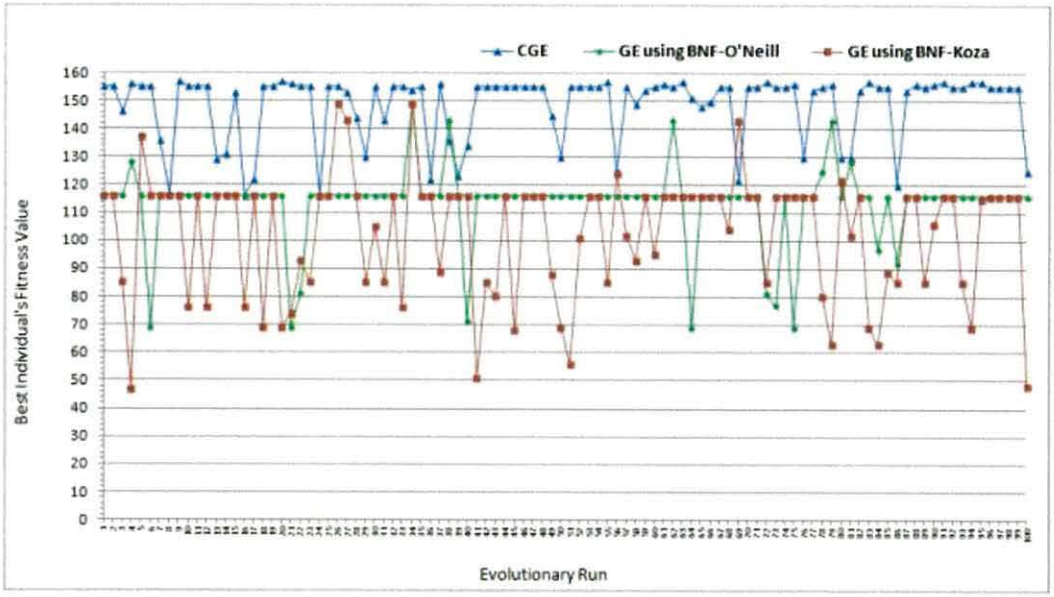


Figure 7.7: Best individuals (ants) per evolutionary run in the Los Altos Hills problem.

The main reasons why Los Altos Hills proved to be so difficult for GE are: first, the two new irregularities introduced which require a more complex behaviour by the ant; and second, because these irregularities first appear at the end of the trail, with the consequence that the evolved population converges to programs tackling only the first part of the trail which is identical to the Santa Fe Trail. In addition, the environment of the Los Altos Hills (100x100) is much larger than this of the Santa Fe Trail (32x32).

Even though the Los Altos Hills is such a difficult and challenging problem, Constituent Grammatical Evolution (CGE) managed to find solutions when standard GE did not. CGE clearly improves the standard Grammatical Evolution algorithm in this problem. Indeed, the best solution in terms of efficiency (number of required steps) found by CGE solves the problem in just 1093 steps. Namely, CGE found a solution much more efficient than the solution mentioned in Koza (1992, p.157) which requires 1,808 steps.

7.6.2 The Hampton Court Maze Problem

The third problem where CGE was benchmarked against GE is the Hampton Court Maze searching problem. The Hampton Court Maze is a simple connected maze of grid size 39 by 23. The objective is to find a computer program to control an artificial traveller (agent), so that it can find the exit at the centre of the maze. The agent starts in the entry point of the maze at the middle bottom facing north. The artificial traveller uses, like in

the Artificial Ant problem, three primitive actions: *move*, *turn right*, and *turn left*. Each of these takes one time unit. In addition, the artificial traveller can use three sensing functions: *wall ahead*, *wall left*, and *wall right* (Sondahl, 2005), each of them requiring no time unit. These sensing functions look into the front, left or right square respectively, and return true if that square contains a wall or false if it is a path.

The interface of the model which was used for the performance of the benchmarking is shown in Figure 7.8. The green square is the entry of the maze and the red square the exit.

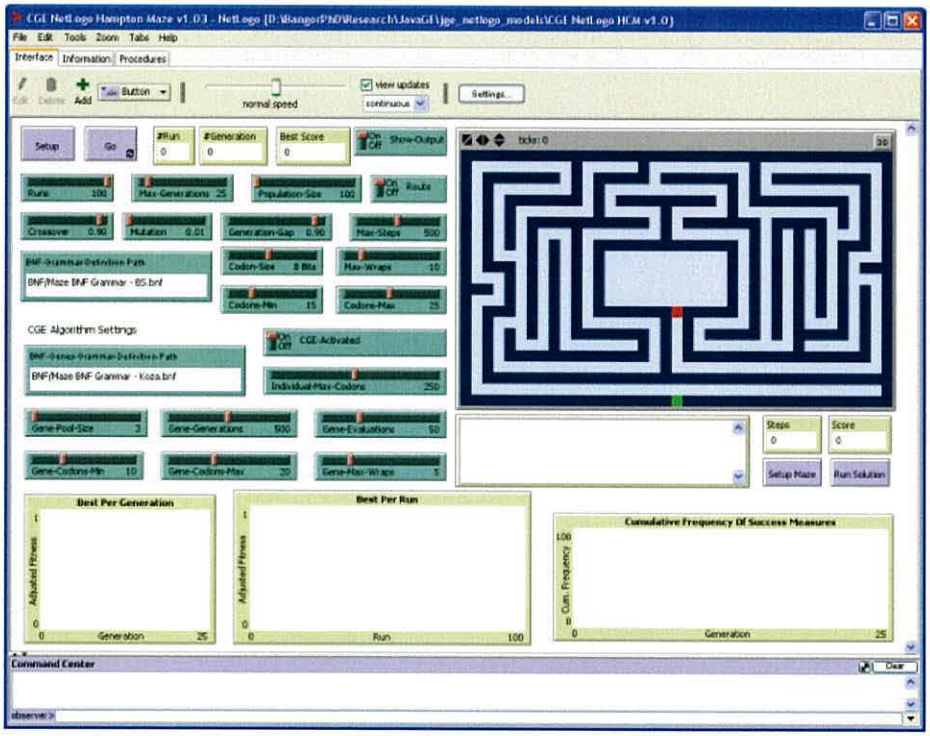


Figure 7.8: Interface of the Evolutionary Hampton Court Maze model for CGE and GE.

As with the Los Altos Hills, three experiments were conducted of one hundred evolutionary runs each in order to compare the performance of CGE, GE using a maze version of BNF-Koza (Listing 7.11), and GE using a maze version of BNF-O'Neill (Listing 7.12). The BNF grammar definitions used in these experiments are variations of those used in the Santa Fe Trail and Los Altos Hills problems with the following differences: the replacement of the food-ahead sensing operator with the *wall-ahead?* operator, the addition of two sensing operators, *wall-left?* and *wall-right?*; and the addition of a non-terminal symbol for choosing any one of these sensing operators when a condition statement is to be selected. The sensing operators were inspired by the work of

Sondahl (2005). The behaviour-switching grammar definition (BNF-BS) used by CGE is shown in Listing 7.13. The experimental configuration is shown in Table 7.10 and Table 7.11.

```
N = {expr, line, condition, op}
T = {turn-left, turn-right, move, ifelse, wall-ahead?, wall-
left?, wall-right?, [, ]}
S = expr
P:
<expr>      ::= <line> |
                <expr> <line>
<line>      ::= ifelse <condition> [ <expr> ][ <expr> ] |
                <op>
<condition> ::= wall-ahead? |
                wall-left? |
                wall-right?
<op>        ::= turn-left |
                turn-right |
                move
```

Listing 7.11: BNF-Koza grammar definition (maze searching version).

```
N = {code, line, condition, <condition-op>, op}
T = {turn-left, turn-right, move, ifelse, wall-ahead?, wall-
left?, wall-right?, [, ]}
S = code
P:
<code>         ::= <line> |
                   <code> <line>
<line>         ::= <condition> |
                   <op>
<condition>    ::= ifelse <condition-op> [ <line> ][ <line> ]
<condition-op> ::= wall-ahead? |
                   wall-left? |
                   wall-right?
<op>           ::= turn-left |
                   turn-right |
                   move
```

Listing 7.12: BNF-O'Neill grammar definition (maze searching version).

```
N = {behaviour, condition, op}
T = {turn-left, turn-right, move, ifelse, wall-ahead?, wall-
left?, wall-right?, [, ]}
S = behaviour
P:
<behaviour> ::= ifelse <condition> [ <op> ][<op> <behaviour>]|
                ifelse <condition> [ <op> ][ <op> ] |
                <op>
<condition> ::= wall-ahead? |
                wall-left? |
                wall-right?
<op>        ::= turn-left |
                turn-right |
                move |
                ... {constituent genes phenotypes}
```

Listing 7.13: BNF-BS grammar definition (maze searching version).

Table 7.10: Grammatical Evolution tableau for Hampton Court Maze.

Objective	Find a computer program in the NetLogo programming language to control an artificial traveller agent so that it can find the centre of the maze.
Terminal Operators	*turn-left, turn-right, move, wall-ahead?, wall-left?, wall-right?*, plus *constituent genes phenotypes* when the CGE algorithm is used instead of the standard GE algorithm.
Raw Fitness	The geometric distance between the agent's position and the entrance to the centre of the maze before the agent times out with 500 operations, divided by the number of new squares of the path visited. Promotes agents approaching the exit by covering greater unexplored distance (traveller).
Standardised Fitness	Same as the raw fitness.
Adjusted Fitness	1 / (1 + Standardised Fitness).
BNF Grammar	CGE: BNF-Koza (maze version) for Genes and BNF-BS (behaviour-switching maze version) for travellers (agents). GE using BNF-Koza: BNF-Koza (maze version). GE using BNF-O'Neill: BNF-O'Neill (maze version).
Evolutionary Algorithm	Steady-State Genetic Algorithm, Generation Gap = 0.9, Selection Mechanism: Roulette-Wheel Selection.
Initial Population	Randomly created with the following restrictions: Minimum Codons = 15 and Maximum Codons = 25.
Parameters	Population Size = 100, Maximum Generations = 25, Mutation Probability = 0.01, Crossover Probability = 0.9, Codon Size = 8, Wraps Limit = 10.

Table 7.11: CGE settings for the Hampton Court Maze problem.

Parameter	Value	Parameter	Value
Codons Limit, *IMC*	250	Gene Evaluations, *E*	50
Gene Pool Size, *S*	3	Gene Codons Min, *CMin*	10
Gene Generations, *G*	500	Gene Codons Max, *CMax*	20
Gene Code Iterations, *L*	10	Gene Max Wraps, *W*	5

The fitness value of the candidate constituent genes is calculated with the formula shown in Equation 7.2

Equation 7.2: Hampton Court Formula 1

$$F = ((V_1 + \ldots + V_E)/E) \times (V_{max}/S)$$

where: F is the fitness value of the constituent gene used for comparison with other genes and for deciding whether it will be placed in the genes pool or not; E is the number of

performed gene's interim evaluations; V_n is the Interim fitness value of the interim evaluation n ($0 \leq n \leq$ E); V_{max} is the maximum interim fitness value found; and S is the number of operations (time steps) performed during the interim evaluation of the gene with the maximum interim fitness. Each interim fitness value of the candidate constituent genes is calculated with the formula shown in Equation 7.3

Equation 7.3: Hampton Court Formula 2

$$V = D_S - (D_P/P)$$

where: V is the interim fitness value; D_S is the geometric distance between the gene and the exit of the maze before the gene executes its code; D_P is the geometric distance between the gene and the exit of the maze after the execution of the gene's code for L times (Gene Code Iterations); and P is the number of new path squares visited during the gene's code execution. This formula promotes genes approaching the exit covering greater unexplored distance.

The results of these experiments are shown in Table 7.12. "Evolutionary Runs" is the number of evolutionary runs performed in the experiment; "Best Solution's Steps", the required steps (agent moves) of the best solution found in terms of efficiency (required steps) in the particular experiment; and "Success Rate", how many evolutionary runs (percentage) found a solution.

Table 7.12: Experimental results for the Hampton Court Maze problem.

	CGE	GE BNF-Koza	GE BNF-O'Neill
Evolutionary Runs	100	100	100
Best Solution's Steps	384	439	494
Success Rate	**82%**	**1%**	**1%**

The best solution found by CGE for the Hampton Court Maze problem (most effective solution in terms of required steps) can be seen in Listing 7.14. It requires just 384 steps to solve the problem from the 500 maximum allowed steps limit.

```
ifelse wall-left?
      [ifelse wall-ahead?
            [turn-right]
            [move]
      ]
      [turn-left move]
```

Listing 7.14: Best solution found by CGE in the Hampton Court Maze problem. It requires 384 steps.

The Hampton Court Maze problem proved to be difficult for Grammatical Evolution, mainly because the first square the agent visits when it enters the maze is assigned a high fitness value due its small geometric distance from the exit. This leads to convergence of the population to this local optimum. Namely it advances individuals which just execute the *move* operator. In contrast, CGE proved to be very effective for this problem because it was able to easily overcome this local optimum due to the constituent genes. Indeed, the best solution it found requires much fewer steps than those found by Grammatical Evolution.

7.6.3 The Chevening House Maze Problem

The second maze searching problem where Constituent Grammatical Evolution was benchmarked against Grammatical Evolution was the Chevening House Maze (see section 2.5.5). The Chevening House Maze is a multiple-connected maze of grid size 47 by 47. Figure 7.9 shows the interface of the NetLogo model used for the benchmarking. The green square is the entry of the maze and the red square the exit.

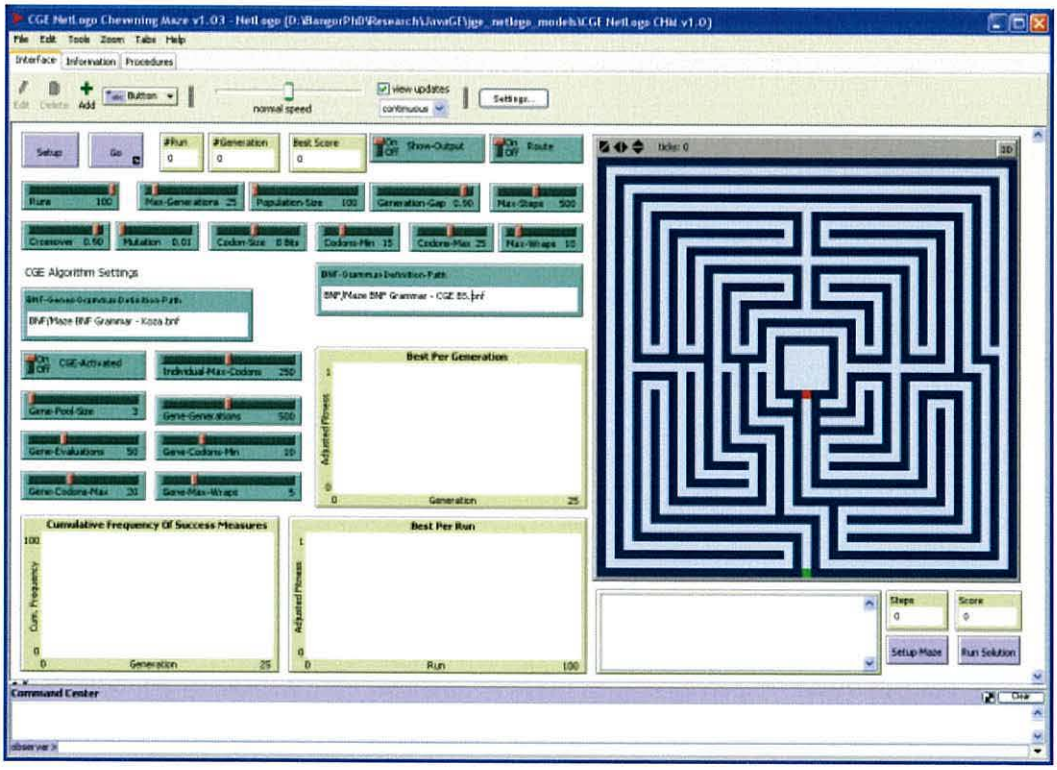


Figure 7.9: Interface of the Evolutionary Chevening House Maze model for CGE and GE.

The same benchmarks as with the Hampton Court Maze were performed with the same experiments setup and configuration of CGE and Grammatical Evolution.

The Table 7.13 shows the results of the experiments. "Evolutionary Runs" is the number of evolutionary runs performed in the experiment; "Best Solution's Steps", the required steps (agent moves) of the best solution found in terms of efficiency (required steps) in the particular experiment; and "Success Rate", how many evolutionary runs (percentage) found a solution. Also, the cumulative frequency measure of success over 100 runs of Constituent Grammatical Evolution and Grammatical Evolution (using BNF-Koza and BNF-O'Neill) can be seen in Figure 7.10.

Table 7.13: Experimental results for the Chevening House Maze problem.

	CGE	GE BNF-Koza	GE BNF-O'Neill
Evolutionary Runs	100	100	100
Best Solution's Steps	314	414	330
Success Rate	**74%**	**6%**	**62%**

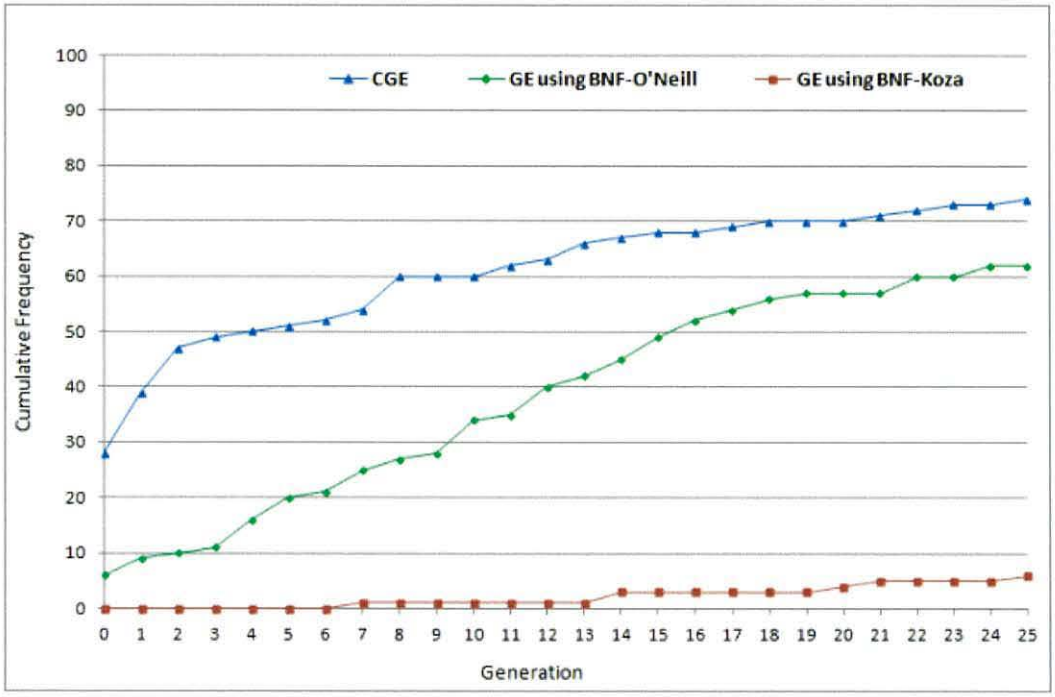


Figure 7.10: Cumulative frequency of success measures over 100 evolutionary runs in the Chevening House Maze problem.

Even though the Chevening House Maze can be considered as a challenging problem compared to other simper types of mazes, Constituent Grammatical Evolution (CGE) has found solutions within 25 generations with a high success rate. Based on the experimental

results, CGE clearly improves the standard Grammatical Evolution algorithm in this problem.

The best solution found by CGE (Listing 7.15) in terms of efficiency (number of required steps), solves the problem in just 314 steps. Namely, CGE found a solution that was much more efficient than the best solutions found by GE using BNF-Koza and GE using BNF-O'Neill which require 414 and 330 steps respectively.

```
ifelse wall-right?
      [ifelse wall-left?
            [move]
            [turn-left move]
      ]
      [ifelse wall-ahead?
            [turn-right move]
            [move]
      ]
```

Listing 7.15: Best solution found by CGE in the Chevening House Maze problem. It requires 314 steps.

It is worthwhile to note here that the best solutions found by CGE and GE in the Chevening House maze require much fewer steps than those found for the Hampton Court maze. The best solutions found by CGE, GE using BNF-O'Neill, and GE using BNF-Koza require 314, 330, and 414 steps respectively. Instead, the best solutions found for the Hampton Court Maze problem by CGE, GE using BNF-O'Neill, and GE using BNF-Koza require 384, 494, and 439 steps respectively. Namely, even though the Chevening House maze is larger than the Hampton Court maze, is a multiply-connected maze rather than a singly-connected maze, and the former contains a larger route from the entry to the exit of the maze than the later, the Chevening House maze seems to be simpler, requiring a strategy of fewer steps to be solved.

Finally, even though, the Chevening House Maze problem seems at a first look to be a more difficult maze to solve than the Hampton Court Maze, mainly because it is larger and apparently more complex, CGE and Grammatical Evolution proved to be able to find solutions with a very high success rate. In the case of Grammatical Evolution, the success rate is much higher in this maze than in the Hampton Court Maze. Namely, GE using BNF-Koza and GE using BNF-O'Neill solve the Chevening House Maze with success rate 6% and 62% respectively, when their success rate in the Hampton Court Maze was just 1%. On the other hand, CGE has a slightly lower success rate in Chevening House maze

against the Hampton Court maze (74% against 82%) which can be explained with the fact that because even though the former maze is larger than the later (a larger environment to be explored), the same maximum limit of 500 steps was used in the experiments. The main reason of the higher success rate of Grammatical Evolution in Chevening House is that it has less deceptive local optima than the Hampton Court maze.

7.7 CGE Statistics in the Santa Fe Trail Problem

7.7.1 Effectiveness of CGE runs

Even though CGE has an overhead of creating and evaluating constituent genes, the statistical results of the experiments conducted with CGE and GE using BNF-Koza on the Santa Fe Trail problem (Table 7.14) reveal that the total number of genes and ants evaluated by CGE is lower than the number of ants evaluated by GE using BNF-Koza.

Table 7.14: CGE and GE comparison in the Santa Fe Trail problem.

CGE and GE(BNF-Koza) Comparison	
Inquiry	**Result**
Total numbers of Grammatical Evolution (GE) experiments and evolutionary runs using the BNF-Koza grammar definition.	5 Experiments 500 Evolutionary Runs
Total numbers of Constituent Grammatical Evolution (CGE) experiments, evolutionary runs and candidate constituent genes (CGE experiments using Gene-Pool-Size 3 and Gene-Generations 850).	5 Experiments 500 Evolutionary Runs 1,275,000 Genes (total)
Amount of CGE candidate constituent genes with a valid phenotype.	532,949 Genes (valid)
Amount of CGE candidate constituent genes with an invalid phenotype.	742,051 Genes (invalid)
Amount of CGE constituent genes (genes used in CGE runs).	1,500 Genes
Success Rate of GE (percentage of evolutionary runs found a solution).	10%
Success Rate of CGE (percentage of evolutionary runs found a solution).	**90%**
Total number of ants over 500 evolutionary runs using GE.	12,322,000 ants (total)
Total number of valid ants over 500 evolutionary runs using GE.	9,646,553 ants (valid)

Total number of invalid ants over 500 evolutionary runs using GE.	2,675,447 ants (invalid)
Total number of ants over 500 evolutionary runs using CGE.	**4,743,500 ants (total)**
Total number of valid ants over 500 evolutionary runs using CGE.	**4,716,243 ants (valid)**
Total number of invalid ants over 500 evolutionary runs using CGE.	**27,257 ants (invalid)**

In a total of 500 evolutionary runs, CGE created 1,275,000 candidate constituent genes and 4,743,500 ants, a total of 6,018,500 genes and ants. Note that 4,716,243 ants and only 532,949 genes had a valid phenotype and were evaluated (because genes are created randomly). Therefore, a total of 5,249,192 valid genes and ants were evaluated by CGE. In contrast, GE using BNF-Koza created more than double the number of ants, namely 12,322,000 of which 9,646,553 were valid and were evaluated. It is readily apparent that CGE requires much less processing power to execute 500 evolutionary runs in the Santa Fe Trail problem than GE using BNF-Koza because the most demanding tasks in these runs is the evaluation of valid genes and ants in the Santa Fe Trail problem which requires initialization and simulation for every gene and ant. It is noted that invalid genes and invalid ants are not evaluated because they are invalid programs due to the incomplete mapping process, therefore they count only in the computational effort regarding the genotype-to-phenotype mapping process which is analysed in section 7.7.4.

The above finding can be easily explained by the fact that even though CGE and GE using BNF-Koza perform the same number of evolutionary runs, the very high effectiveness of CGE (90% against 10%) implies that the generations of populations created is far less than these created by GE using BNF-Koza. That is because when a solution is found, the evolutionary run terminates and does not reach the maximum number of generations as is the case when no solution is found. Also, with CGE the invalid ants are far less than those of standard GE because it requires smaller genotypes to complete the phenotype of an individual due to the BNF-BS bias and the constituent genes.

7.7.2 Efficiency of CGE Solutions

Regarding the efficiency of the solutions found by CGE and Grammatical Evolution in the Santa Fe Trail problem, CGE seems to improve Grammatical Evolution not only on the very best solution found in all experiments, but in the whole set of solutions found by

these algorithms in all evolutionary runs of the experiments. Namely, the overall quality of the solutions found by CGE is higher than these found by Grammatical Evolution.

The graph in Figure 7.11 shows for each algorithm (CGE and GE using BNF-Koza) the distribution of the solutions found, with regards to the required steps performed by the ant. The graph depicts that 68% of the solutions found by CGE perform moves in a range of 401-450 steps in order to find all 89 pieces of food in the trail. Instead, the 52% of the solutions found by GE using BNF-Koza requires moves in a range of 601-650 steps.

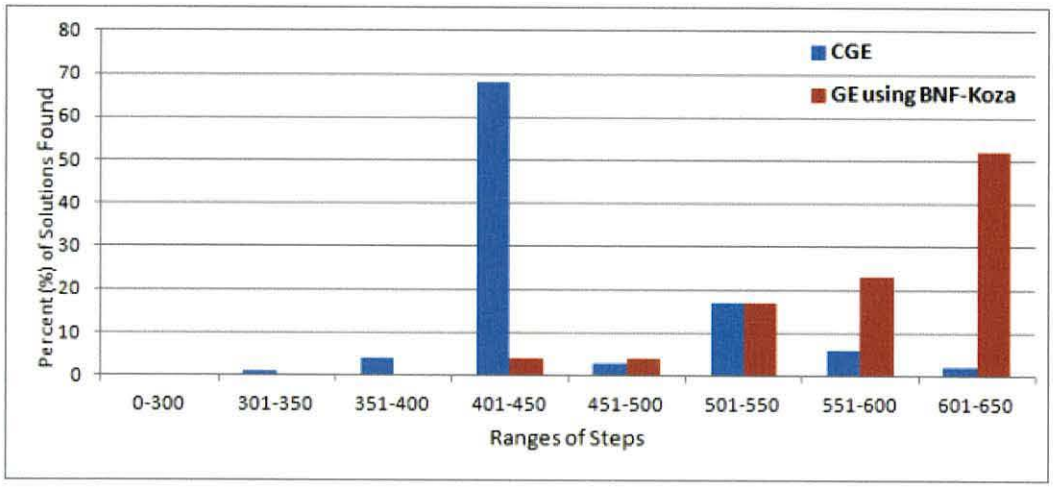


Figure 7.11: CGE vs. GE – Percentages of solutions found for varying ranges of steps.

The graph in Figure 7.12 shows for each algorithm the distribution of the solutions found, with regards to the size of the genotype in bits.

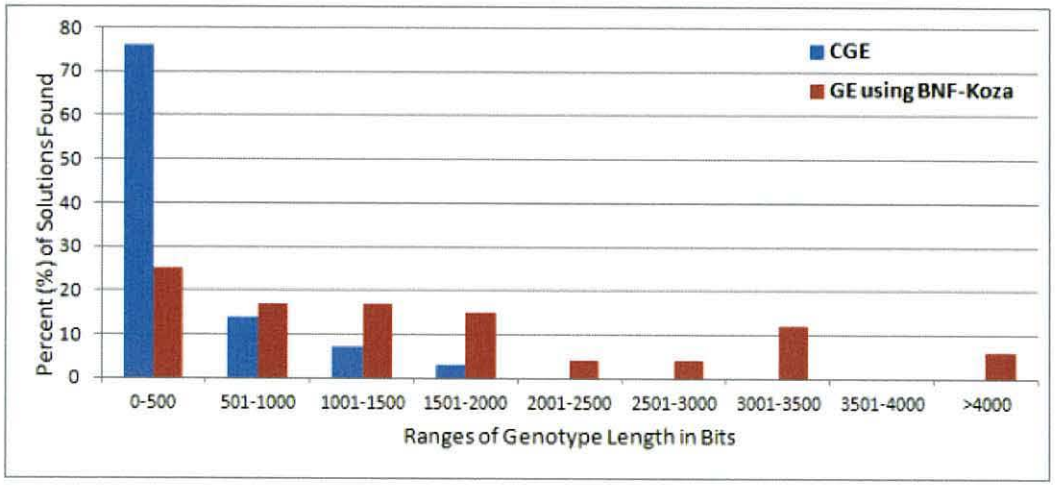


Figure 7.12: CGE vs. GE - Percentages of solutions found per genotype size ranges.

What is interesting in this graph is the observation that the solutions found by CGE require smaller genotypes than these found by GE using BNF-Koza. Namely, the 76% of the solutions found by CGE are created with genotypes of size less than 500 bits and no solution had a genotype larger than 2000 bits. Additionally to the genotype size maximum limit, this is because constituent genes compress more information into a single choice of a production and the language bias toward conditional statements of the BNF-BS grammar definition biases toward formation of larger structures, consequently there are less codon reads during the mapping process and smaller genotypes are required.

Also, it is observed here that imposing a limit of 250 codons (2,000 bits) to the genotype of an individual, as is the case with the CGE experiments conducted so far (except in Los Altos Hills where the limit was 750 codons), actually does not negatively affect the success rate of the CGE algorithm. That is because almost all solutions found in the conducted experiments require far less genotype size than this limit and consequently it does not seem probable that a solution of larger genotype would be found by CGE during a run which would be rejected because its size exceeded the max codons limit (at least not in a magnitude that could affect in practice the success rate).

7.7.3 Best Solutions Phenotypes

This section provides a listing of the phenotypes of the best solutions found by GE using BNF-Koza, GE using BNF-O'Neill, and CGE respectively in the experiments conducted in the Santa Fe Trail problem. The best solutions have been chosen according to three different criteria: the number of steps, the size of the genotype, and the size of the phenotype.

Number of Steps

The phenotypes (in NetLogo) of the best solutions found in terms of ant's steps in the conducted experiments are shown in Listing 7.16, Listing 7.17, and Listing 7.18.

```
ifelse food-ahead
  [move]
  [ifelse food-ahead
    [turn-right]
    [turn-left
     ifelse food-ahead
```

```
      [move move turn-left turn-right]
      [turn-right]
    turn-right
    ifelse food-ahead
      [move]
      [ifelse food-ahead
        [turn-right]
        [turn-left]
       move
      ]
    ]
  ]
```

Listing 7.16: Best solution found by GE using BNF-Koza (415 steps).

```
ifelse food-ahead
  [move]
  [turn-right]
ifelse food-ahead
  [move]
  [turn-right]
move
turn-right
ifelse food-ahead
  [turn-left]
  [ifelse food-ahead
    [turn-right]
    [ifelse food-ahead
      [ifelse food-ahead
        [move]
        [turn-left]
      ]
      [ifelse food-ahead
        [move]
        [ifelse food-ahead
          [turn-right]
          [ifelse food-ahead
            [turn-right]
            [ifelse food-ahead
              [turn-left]
              [ifelse food-ahead
                [move]
                [ifelse food-ahead
                  [ifelse food-ahead
                    [move]
                    [turn-right]
                  ]
                  [turn-right]
                ]
              ]
            ]
          ]
        ]
      ]
    ]
  ]
```

Listing 7.17: Best solution found by GE using BNF-O'Neill (607 steps).

```
ifelse food-ahead
  [move]
  [turn-right
   ifelse food-ahead
     [ifelse food-ahead
       [ifelse food-ahead
         [move move]
         [move]
        ifelse food-ahead
          [move move]
          [move]
       ]
       [turn-left]
      move
     ]
     [ifelse food-ahead
        [move]
        [turn-right]
      ifelse food-ahead
        [turn-left]
        [turn-right
         ifelse food-ahead
           [move]
           [ifelse food-ahead
             [move]
             [turn-right]
            move
           ]
        ]
     ]
  ]
```

Listing 7.18: Best solution found by CGE (337 steps).

Genotype Size

The genotypes and the phenotypes (in NetLogo) of the best solutions found in terms of genotype size (bits) in the conducted experiments are shown in Listing 7.19, Listing 7.20, and Listing 7.21.

```
Genotype:
01100001110010110000010100001101101000001101010011001100010001010
11010101011101000000001110010111111001000101001011111011111001100
11111100101011

Phenotype:
ifelse food-ahead
  [turn-left]
  [turn-right]
ifelse food-ahead
  [move]
  [turn-right]
Move
```

```
turn-right
ifelse food-ahead
  [turn-left]
  [turn-right]
```

Listing 7.19: Shortest genotype found by GE using BNF-Koza (142 bits, 617 steps).

```
Genotype:
10111101000100111001111110001011000110001010010110001100011110
10010001000001010111111100

Phenotype:
move
turn-right
ifelse food-ahead
  [turn-left]
  [turn-right]
turn-right
ifelse food-ahead
  [move]
  [turn-right]
```

Listing 7.20: Shortest genotype found by GE using BNF-O'Neill (90 bits, 615 steps).

```
Genotype:
00001001000001000011101011101101010001001110011101000101000000
1011001000

Phenotype:
ifelse food-ahead
  [ifelse food-ahead
    [move]
    [turn-right]
  ]
  [ifelse food-ahead
    [move]
    [turn-right]
   ifelse food-ahead
    [move]
    [ifelse food-ahead
      [move]
      [turn-right]
     ifelse food-ahead
      [ifelse food-ahead
        [move]
        [turn-right]
      ]
      [ifelse food-ahead
        [move]
        [turn-right]
       ifelse food-ahead
        [ifelse food-ahead
          [move]
          [ turn-right]
        ]
```

```
          [ifelse food-ahead
            [move]
            [turn-right]
           ifelse food-ahead
             [ifelse food-ahead
               [move]
               [turn-right]
             ]
             [move]
          ]
       ]
    ]
  ]
```

Listing 7.21: Shortest genotype found by CGE (73 bits, 405 steps).

Phenotype Size

The phenotypes of the best solutions found in terms of phenotype size (operators) in the conducted experiments are shown in Listing 7.22, Listing 7.23, and Listing 7.24.

```
ifelse food-ahead
  [move]
  [turn-left
   turn-left
   ifelse food-ahead
     [move]
     [turn-right]
  ]
move
turn-right
```

Listing 7.22: Shortest phenotype found by GE using BNF-Koza (9 operators, 589 steps).

```
ifelse food-ahead
  [move]
  [turn-right]
move
turn-right
ifelse food-ahead
  [turn-right]
  [turn-left]
turn-left
```

Listing 7.23: Shortest phenotype found by GE using BNF-O'Neill (9 operators, 611 steps).

```
ifelse food-ahead
  [turn-left]
  [turn-right
   ifelse food-ahead
```

```
    [move]
    [turn-right
     ifelse food-ahead
       [turn-right]
       [turn-right
        ifelse food-ahead
          [move]
          [turn-right]
        move
       ]
    ]
]
```

Listing 7.24: Shortest phenotype found by CGE (13 operators, 519 steps).

As can be observed in these solutions, CGE was able to find, as already mentioned, not only the solution with the fewer steps and the solution with the smallest genotype, but also the steps required by the solution with the smallest genotype are less than these of the corresponding solutions found by GE using BNF-Koza and GE using BNF-O'Neill.

7.7.4 Processing Requirements of CGE

In order to evaluate the performance (in terms of required processing and execution time – duration) of the CGE algorithm in comparison with the standard Grammatical Evolution algorithm in the Santa Fe Trail problem, the NetLogo model used in these experiments logs additional information about the elements of interest.

The retrieval of time information has been implemented with the native NetLogo reporter "timer". According to the Dictionary of the NetLogo Manual (Wilensky, 1999), the "timer" procedure reports how many seconds have passed since the command reset-timer was last run (or since NetLogo started). The potential resolution of the clock is milliseconds. Whether, the resolution is that high in practice may vary from system to system, depending on the capabilities of the underlying Java Virtual Machine.

The following information are retrieved and logged during the execution of an experiment in the Santa Fe Trail problem:

- Total number of genotype-to-phenotype mappings during an experiment (each experiment performs 100 evolutionary runs).
- Number of mappings, during an experiment of 100 evolutionary runs, with execution time greater than or equal to 1 millisecond.

- Total execution time (in seconds) of mappings with execution time greater than or equal to 1 millisecond, during an experiment of 100 evolutionary runs. The resolution of the clock in NetLogo is in milliseconds but most mappings require less time. Consequently, the total execution time of mappings takes in account only the mappings with duration greater than or equal to 1 millisecond. For all other mappings, NetLogo (reporter "time") returns execution time zero.
- Number of generations created in each evolutionary run (the initial generation 0 is also counted).
- Time (in seconds) required for the creation of the constituent genes pool of each evolutionary run.
- Execution time (in seconds) of each GE evolutionary run. This time plus the time right above for the pool creation, results in the execution time of the CGE evolutionary run ($T_{CGE} = T_{GE} + T_{CGP}$).

For the collection and analysis of the above information, two experiments have been conducted of 100 evolutionary runs each one: CGE and GE using BNF-Koza, both using the setup and configuration mentioned in section 7.5.1. All experiments were performed in a personal desktop computer with Windows XP (SP3), Java SE 6, and NetLogo 4.1.1 with the following hardware configuration: Intel Core 2 Duo CPU, E8400 @ 3.00GHz, 3.00 GHz, 3.25GB of RAM.

The statistical results which have been calculated from the logged time information of the experiments performed are shown in Table 7.15.

Table 7.15: Processing statistics of CGE and GE using BNF-Koza.

Statistic / Metric	GE using BNF-Koza	CGE
Mappings Total	2,441,000	**861,500**
Mappings with ET>0	175,004	**1,268**
Percentage of Mappings with ET>0	7.17%	**0.15%**
Mappings Execution Time (sec)	18,293.05	**20.44**
Mappings Time Percentage	52.75%	**0.17%**
Average Mapping Duration (sec)	0.10453	**0.01612**
Normalised Average Mapping Duration (sec)	0.00796	**0.00052**
Generations Total	4,882	**1,723**
Generations per Run	49	**17**

Experiment Duration (sec)	34,676	**12,075**
Pools Creation Duration (sec)	**0**	2,939
Pools Creation Percentage	**0%**	24%
Average Run Duration (sec)	347	**121**
Average Pool Creation Duration (sec)	**0**	29
Average Pool Creation Percentage	**0%**	24%
Average Generation Duration (sec)	7.10	**7.01**

"Mappings Total" is the total number of genotype-to-phenotype mappings performed in the experiment; "Mappings with ET>0" is the number of mappings with execution time equal to or greater than 1 millisecond; "Percentage of Mappings with ET>0" is the percentage of mappings with execution time equal to or greater than 1 millisecond over the total mappings performed in the experiment; "Mappings Execution Time (sec)" is the execution time (duration) in seconds of all mappings with execution time equal to or greater than 1 millisecond; "Mappings Time Percentage" is the percentage of the measured mappings execution time over the execution time of whole the evolutionary run; "Average Mapping Duration (sec)" is the average duration in seconds of a genotype-to-phenotype mapping (only mappings with duration equal to or greater than 1 millisecond are taken in account); In "Normalised Average Mapping Duration (sec)" all mappings are taken in account by making the assumption that mappings with execution time less than 1 millisecond have an average duration of 0.5 milliseconds; "Generations Total" is the number of all generations created during the experiment (100 evolutionary runs); "Generations per Run" is the average number of generations created during an evolutionary run of the experiment; "Experiment Duration (sec)" is the duration in seconds of the experiment; "Pools Creation Duration (sec)" is the execution time (duration) in seconds of the creation of the constituent genes pools; "Pool Creation Percentage" is the percentage of the pools creation duration over the duration of whole the evolutionary run; "Average Run Duration (sec)" is the average duration in seconds of an evolutionary run of the experiment (includes constitute genes pool creation execution time and standard grammatical evolution execution time); "Average Pool Creation Duration (sec)" is the average duration in seconds of the constituent genes pool of an evolutionary run of the experiment; "Average Pool Creation Percentage" is the percentage of the average duration of the creation of the constituent genes pool over the average execution time of whole an evolutionary run of the experiment; and "Average Generation Duration

(sec)" is the average execution time (duration) in seconds of a generation creation and evaluation. The last is calculated as follows: *Average Generation Duration = Experiment Duration / Generations Total.*

The results in Table 7.15 show that Constituent Grammatical Evolution performs faster than GE using BNF-Koza in the Santa Fe Trail problem. The difference in the total duration of an experiment is mainly the result of the difference in the number of mappings ("Mappings Total") because in CGE fewer generations of ants are required due to its higher success rate (as discussed in section 7.7.1). But this is not a fair metric, so the average duration time of the creation and evaluation of a population generation has been calculated where, as can be seen in the results, CGE is faster than GE using BNF-Koza as well. Namely, the average duration for the creation and evaluation of a generation is 7.01 seconds in CGE, against 7.10 seconds in GE using BNF-Koza.

Even though CGE requires substantial additional processing power before each Grammatical Evolution run, because of the creation and evaluation of the candidate constituent genes (its duration is the 24% of the total duration of an evolutionary run), it achieves a lower execution time per generation than GE using BNF-Koza in the specific setup, because CGE results to much smaller genotypes than GE, consequently faster genotype-to-phenotype mappings of an individual (namely less processing) are required as shown in Table 7.15. Section 7.8.3 demonstrates the impact of the genotype-to-phenotype mapping process in the computation effort required by Grammatical Evolutions and shows the dramatic improvement that can be achieved when the genotype size is decreased.

7.8 Analysis of CGE

The promising results of the experiments naturally raises the question what is the reason of the high performance of CGE compared to standard Grammatical Evolution and to what extent each of the three unique features of CGE – constituent genes, behaviour-switching approach, and genotype size max limit – affect the effectiveness and efficiency of the algorithm. A series of experiments are conducted and statistical results are provided in this section in order to highlight the contribution of each of these features in the performance of CGE, to reveal the nature of the constituent genes, and finally to discuss the reasons of the observed performance of Constituent Grammatical Evolution.

7.8.1 The Constituent Genes Feature

In order to evaluate the constituent genes feature in isolation, the BNF-BS grammar is replaced by the original (unbiased) grammars which are used by Grammatical Evolution when it is benchmarked on the Santa Fe Trail, Los Altos Hills, Hampton Court Maze and Chevening House Maze problems in the experiments of the previous sections. Therefore, CGE is configured in the setups of this section to use one grammar definition for both the constituent genes and the individuals and without forcing a genotype size limit.

In particular, for the Santa Fe Trail problem, the setup of the experiment is the same with these conducted to benchmark CGE against Grammatical Evolution in section 7.5 (see Table 7.1 and Table 7.2) with the following variation: CGE uses the BNF-Koza grammar definition (Listing 5.2) for the individuals instead of the BNF-BS (Listing 7.8) in order to rule-out the bias imposed by this grammar, and also it does not enforce a limit to the size of the genotype of the individuals. The constituent genes are added to the BNF-Koza grammar definition in the same way as in the BNF-BS (namely, as productions in the <*op*> non-terminal symbol).

The same approach for the benchmarking of the constituent genes feature is followed for the other problems as well. Namely, in the Los Altos Hills, the setup of Table 7.7 and Table 7.8 is used with the replacement of the BNF-BS grammar of Listing 7.10 with the BNF-Koza grammar (Listing 5.2). In the maze searching problems, the maze version of the BNF-Koza grammar definition (Listing 7.11) is used for both the agents and the genes using the configuration of Table 7.10 and Table 7.11. Namely, the bias imposed in CGE by the grammar BNF-BS of Listing 7.13 and the genome max limit are ruled out in these experiments as well.

In each experiment, 100 evolutionary runs were performed and the experimental results are shown in Table 7.16. "Evolutionary Runs" is the number of evolutionary runs performed in the experiment; "Best Solution's Steps", the required steps (ant moves) of the best solution found in terms of efficiency (required steps) in the particular experiment; and "Success Rate", how many evolutionary runs (percentage) found a solution.

Table 7.16: Constituent genes benchmark results on Santa Fe Trail, Los Altos Hills, Hampton Court Maze, and Chevening House Maze using BNF-Koza.

	Santa Fe Trail	Los Altos Hills	Hampton Court Maze	Chevening House Maze
Evolutionary Runs	100	100	100	100
Best Solution's Steps	493	--	384	330
Success Rate	**37%**	**0%**	**15%**	**26%**

The experimental results show that the constituent genes feature improves the success rate of Grammatical Evolution and this is due to the advantages of modularity (reusable building blocks of code). Namely, from 10% success rate to 37% in the Santa Fe Trail problem, from 1% to 15% in the Hampton Court Maze, and from 6% to 26% in the Chevening House Maze problem. Instead, in the Loss Altos Hills problem, neither approach managed to find a solution which makes this problem a challenging case for future investigation and experimentation.

Listing 7.25 shows a sample genes pool which was created for an evolutionary run of the Santa Fe Trail experiment of this section (see Table 7.16) and Listing 7.26 shows the grammar definition – after the addition of the constituent genes of the pool as productions (see lines 1, 2 and 3) – which was used by Grammatical Evolution in this run.

```
Constituent gene (1)
ifelse food-ahead [ move ][ turn-right ]

Constituent gene (2)
ifelse food-ahead
               [ move ]
               [ turn-right ifelse food-ahead
                                [ move ]
                                [ turn-right ] ] |
Constituent gene (3)
move ifelse food-ahead [ move ][ turn-right ]
```

Listing 7.25: Sample of a genes pool from the Santa Fe Trail experiment containing three constituent genes phenotypes.

```
<expr> ::= <line> |
           <expr> <line>
<line> ::= ifelse food-ahead [ <expr> ][ <expr> ] |
           <op>
<op>   ::= turn-left |
           turn-right |
```

```
move |
ifelse food-ahead [ move ][ turn-right ] |      (1)
ifelse food-ahead                               (2)
    [ move ]
    [ turn-right ifelse food-ahead
                    [ move ]
                    [ turn-right ] ] |
move ifelse food-ahead [ move ][ turn-right ]   (3)
```

Listing 7.26: Sample of a grammar definition from the Santa Fe Trail experiment after the addition of the phenotypes of the constituent genes of the genes pool of Listing 7.25 before the start of a Grammatical Evolution run.

Regarding the form of the candidate constituent genes (which are created randomly) and the form of the constituent genes of the pools (which are evaluated and selected based on the fitness function of Equation 7.1), the most common of them are provided below. The statistics are based on a sample of 1000 evolutionary runs on the Santa Fe Trail problem. Each pool of constituent genes is created with the parameters of Table 7.2. A sample of 1000 runs is chosen in order to provide reliable figures because the candidate constituent genes are created randomly.

Table 7.17 displays the 10 most common randomly created candidate constituent genes and their occurrence percentage over the total number of created candidates with a valid phenotype (complete genotype-to-phenotype mapping). Table 7.18 shows the 10 most common constituent genes in the created gene pools and their occurrence percentage over the total number of genes in pools, namely, the most common phenotypes (building blocks) which were added to the original grammar definition as productions.

Table 7.17: Ten most common randomly created candidate constituent genes and their occurrence percentage over the total number of valid candidates.

Rank	Candidate Constituent Genes	Occurrence
1	turn-left	20.06%
2	move	19.88%
3	turn-right	19.84%
4	turn-left turn-left	1.71%
5	turn-right turn-left	1.67%
6	move turn-left	1.67%
7	turn-left move	1.66%

8	turn-left turn-right	1.66%
9	turn-right turn-right	1.66%
10	move move	1.65%

Table 7.18: Ten most common constituent genes in the created gene pools and their occurrence percentage over the total number of genes in pools.

Rank	Constituent Genes in Pool	Occurrence
1	ifelse food-ahead [move][turn-right]	36.53%
2	ifelse food-ahead [move][turn-left]	32.37%
3	move	7.03%
4	move move	3.90%
5	move ifelse food-ahead [move][turn-right]	0.93%
6	ifelse food-ahead [ifelse food-ahead [move][turn-right]][turn-right]	0.90%
7	ifelse food-ahead [ifelse food-ahead [move][turn-left]][turn-right]	0.80%
8	ifelse food-ahead [ifelse food-ahead [move][turn-left]][turn-left]	0.80%
9	ifelse food-ahead [ifelse food-ahead [move][move]][turn-right]	0.77%
10	ifelse food-ahead [move][ifelse food-ahead [turn-left][turn-right]]	0.70%

7.8.2 The Behaviour-Switching Feature

For the evaluation of the behaviour-switching approach in isolation, a series of experiments is conducted where CGE is configured to rule-out the constituent genes and genotype size max limit features and to use a grammar which bias the search toward conditional statements in the start of the control program (as BNF-BS does) and defines a search space semantically equivalent with this defined by the original grammars (BNF-Koza artificial and maze versions). Therefore CGE becomes a standard Grammatical Evolution algorithm which uses a particular declarative search bias (behaviour-switching approach) stated by a modified version of the original grammar definitions.

In particular, the standard Grammatical Evolution algorithm is configured with the setup of Table 7.1 using the BNF grammar definition shown in Listing 7.27 for the Santa Fe Trail and the setup of Table 7.7 and grammar definition in Listing 7.28 f for the Los Altos Hills. The productions of the non-terminal <*op*> with the phenotypes of the constituent

genes in BNF-BS are replaced here by the non-terminal *<complex-op>* with the same probability to be selected (50%) as a constituent gene in the configuration of the CGE experiments with genes pool size 3.

```
<behaviour>  ::= ifelse food-ahead [<op>][<op> <behaviour>] |
                 ifelse food-ahead [ <op> ][ <op> ] |
                 <op>
<op>         ::= <single-op>  |
                 <complex-op>
<complex-op> ::= <line> |
                 <complex-op> <line>
<line> ::= ifelse food-ahead [ <complex-op> ][ <complex-op>] |
           <single-op>
<single-op>  ::= turn-left |
                 turn-right |
                 move
```

Listing 7.27: BNF-Koza grammar definition for the Santa Fe Trail problem with a declarative search bias toward conditional statements in the start of the program.

```
<behaviour> ::=
       <behaviour> ifelse food-ahead [<op>][<op> <behaviour>] |
       <behaviour> ifelse food-ahead [<op>][<op> <behaviour>] |
       ifelse food-ahead [ <op> ][ <op> <behaviour> ] |
       ifelse food-ahead [ <op> ][ <op> <behaviour> ] |
       ifelse food-ahead [ <op> ][ <op> ] |
       <op>
<op>         ::= <single-op>  |
                 <complex-op>
<complex-op> ::= <line> |
                 <complex-op> <line>
<line> ::= ifelse food-ahead [ <complex-op> ][ <complex-op>] |
           <single-op>
<single-op>  ::= turn-left |
                 turn-right |
                 move
```

Listing 7.28: BNF-Koza grammar definition for the Los Altos Hills problem with a declarative search bias toward conditional statements in the start of the program (the bias toward conditional statements is similar to the bias used in the CGE experiment on the same problem; see Listing 7.10).

In the same way, Grammatical Evolution algorithm is configured for the maze searching problems with the setup of Table 7.10 and using the grammar definition of Listing 7.29.

```
<behaviour>  ::= ifelse <condition> [<op>][<op> <behaviour>] |
                 ifelse <condition> [ <op> ][ <op> ] |
                 <op>
<op>         ::= <single-op>  |
```

```
                    <complex-op>
<complex-op> ::= <line> |
                    <complex-op> <line>
<line> ::= ifelse <condition> [<complex-op>][<complex-op>] |
           <single-op>
<condition>  ::= wall-ahead? |
                    wall-left? |
                    wall-right?
<single-op>  ::= turn-left |
                    turn-right |
                    move
```

Listing 7.29: BNF-Koza maze version grammar definition for the Hampton Court Maze and Chevening House Maze problems with a declarative search bias toward conditional statements in the start of the program.

In each experiment, 100 evolutionary runs were performed and the experimental results are shown in Table 7.19. "Evolutionary Runs" is the number of evolutionary runs performed in the experiment; "Best Solution's Steps", the required steps (ant moves) of the best solution found in terms of efficiency (required steps) in the particular experiment; and "Success Rate", how many evolutionary runs (percentage) found a solution.

Table 7.19: Behaviour-switching benchmark results on Santa Fe Trail, Los Altos Hills, Hampton Court Maze, and Chevening House Maze.

	Santa Fe Trail	Los Altos Hills	Hampton Court Maze	Chevening House Maze
Evolutionary Runs	100	100	100	100
Best Solution's Steps	377	--	--	414
Success Rate	**37%**	**0%**	**0%**	**5%**

As shown by the results, when Grammatical Evolution uses the feature of the conditional behaviour-switching, a success rate improvement is achieved in the Santa Fe Trail problem from 10% to 37%. Instead, in the other problems there is neither an increase or decrease in performance where Grammatical Evolution using BNF-Koza displayed 0% success rate on the Los Altos Hills, 1% on Hampton Court Maze, and 6% on the Chevening House Maze.

7.8.3 The Genotype Maximum Size Limit Feature

The genotype maximum size limit feature of CGE using the value of the setups in Table 7.2 and Table 7.11 (*IMC* value 250 codons) is isolated and evaluated in the artificial ant and maze searching problems using the standard Grammatical Evolution algorithm in

Santa Fe Trail and Chevening House Maze respectively while enforcing a limit in the genotype size of the same value.

Namely, two experiments have been conducted with Grammatical Evolution using BNF-Koza (Listing 5.2) for the Santa Fe Trail problem and BNF-Koza maze version (Listing 7.11) for the Chevening House Maze problem. GE is configured with the setups of Table 7.1 and Table 7.10 respectively (same setups where GE has been benchmarked on these problems in previous sections) in addition with the enforcement of a genotype size maximum limit (*IMC*) of value 250 codons.

In each experiment, 100 evolutionary runs were performed and the results are shown in Table 7.20. "Evolutionary Runs" is the number of evolutionary runs performed in the experiment; "Best Solution's Steps", the required steps (ant moves) of the best solution found in terms of efficiency (required steps) in the particular experiment; and "Success Rate", how many evolutionary runs (percentage) found a solution.

Table 7.20: Grammatical Evolution using BNF-Koza (artificial ant and maze searching versions) and genotype size limit 250 codons in SFT and CHM problems.

	Santa Fe Trail	Chevening House Maze
Evolutionary Runs	100	100
Best Solution's Steps	387	369
Success Rate	**11%**	**5%**

The results show that the size limit value of 250 codons which is used in these and the CGE experiments has no practical effect on the success rate of Grammatical Evolution using BNF-Koza (artificial ant and maze search versions) on the Santa Fe Trail and Chevening House Maze problems where it displays 10% (Table 7.4) and 6% (Table 7.13) success rate respectively.

In order to confirm that using the limit of 250 codons there is no practical effect in the success rate of the examined problems, another series of experiments is conducted on the Santa Fe Trail problem, this time using CGE with the setup of Table 7.1 and Table 7.2 with the only difference being that the genome size limit (*IMC*) is discarded. Table 7.21 shows the results of these experiments. The column "Best" shows the best value of all five experiments. "Runs" is the number of evolutionary runs performed in each experiment, "Steps", the required steps of the best solution found in the particular experiment,

"Success", how many evolutionary runs (percentage) found a solution, and "Avg. Success", the average success rate of all five experiments.

Table 7.21: CGE on SFT without genotype size maximum limit.

	Exp #1	Exp #2	Exp #3	Exp #4	Exp #5	**Best**
Runs	100	100	100	100	100	**100**
Steps	387	397	373	385	399	**373**
Success	93%	93%	89%	84%	85%	**93%**
Avg. Success	**89%**					

The results confirm the previous observation that using a genotype limit with value 250 codons has no practical effect on the success rate on problems of a magnitude (required effective genome) similar to that of the problems examined in this work. This is probably because a much smaller genotype size than the limit is required by Grammatical Evolution to be effective. Indeed, in CGE it seems that the genotype limit has no practical effect neither in the solution quality (see Table 7.3). Consequently, the questions that arise from the above results are whether there is a lower threshold in the genome size limit that affects the performance of Grammatical Evolution (therefore of CGE as well, but taking in account a necessary adjustment of the limit because the later requires genomes of smaller size due to behaviour-switching and constituent genes) and what is the impact of this feature in the computational effort of Grammatical Evolution.

To answer these questions, a series of Grammatical Evolution using BNF-Koza experiments in the Santa Fe Trail problem were conducted with and without genome size limits of various sizes and measuring the impact of these limits on the volume (non-terminal symbols expansions) and total execution time of the genotype-to-phenotype mappings. The experimental results are compared with the previously experiment conducted on Santa Fe Trail using Grammatical Evolution with limit value 250 codons. All experiments were performed in a laptop computer with Windows 7 Home Premium 64-bit (SP1), Java SE 6, and NetLogo 4.1.3 with the following hardware configuration: Intel Core i5-2430M (dual core) CPU @ 2.4GHz, 2.4GHz, 6 GB of RAM.

The results are shown in Table 7.22. "Evolutionary Runs" is the number of evolutionary runs performed in each experiment; "Best Solution Steps", the required steps (ant moves) of the best solution found in terms of efficiency (required steps) in the particular

experiment; "Best Solution Length (bits)", the size of the genotype in bits of this solution; "Total Ants", the total number of ants of all generations of the particular experiment; "Invalid Ants", how many ants of the total ants had an invalid phenotype (incomplete genotype-to-phenotype mapping); "Non-Terminal Expansions", the total number of non-terminal symbols that expanded to one of its productions during all the genotype-to-phenotype mappings in the experiment (namely, how many times a codon was read to expand a non-terminal during the creation of the derivation trees of the individuals); "Mappings Total Time (sec)", the required total execution time in seconds for the performance of these non-terminal expansions and therefore of the genotype-to-phenotype mappings (both complete and incomplete); and "Success Rate", how many evolutionary runs (percentage) found a solution.

Table 7.22: Results in the Santa Fe Trail problem of Grammatical Evolution using BNF-Koza and applying various genotype size limits (IMC).

	Limit 25	Limit 50	Limit 150	Limit 250	No Limit
Evolutionary Runs	100	100	100	100	100
Best Solution Steps	631	507	387	387	497
Best Solution Length (bits)	200	330	1200	2000	748
Total Ants	2,540,000	2,467,000	2,468,500	2,461,000	2,459,500
Invalid Ants	1,055,615	824,246	597,067	547,791	578,660
Non-Terminal Expansions	247,239,675	306,436,366	436,637,071	529,792,679	673,426,213
Mappings Total Time (sec)	**207.00**	**403.00**	**2,092.00**	**4,505.00**	**16,014.00**
Success Rate	**1%**	**11%**	**13%**	**11%**	**12%**

The results show that with genotype size limit 25 codons, the success rate of Grammatical Evolution drops dramatically. From the limit of 50 codons and above, there is no practical effect in the success rate and solution quality. Regarding the later, no safe conclusions can be made due to the small sample of solutions found. Instead, an impact is observed on the volume of the genotype-to-phenotype mappings (number of expansions of non-terminal symbols to their productions) and the computational effort (time) they require. Enforcing the genotype size limit, a significant reduction is displayed in the volume (non-terminal expansions) of the performed mappings and their total execution time. Also, it is observed that the increase of the number of non-terminal expansions is not linear with the increase of the required execution time. This can be explained because the computational effort

which is required for the expansion of a non-terminal to one of its productions depends also on the size of the derivation tree (phenotype) that is created and the position of the non-terminal to be expanded in this deviation tree (these are dependent on the specific implementation of the mapping algorithm in a computer program).

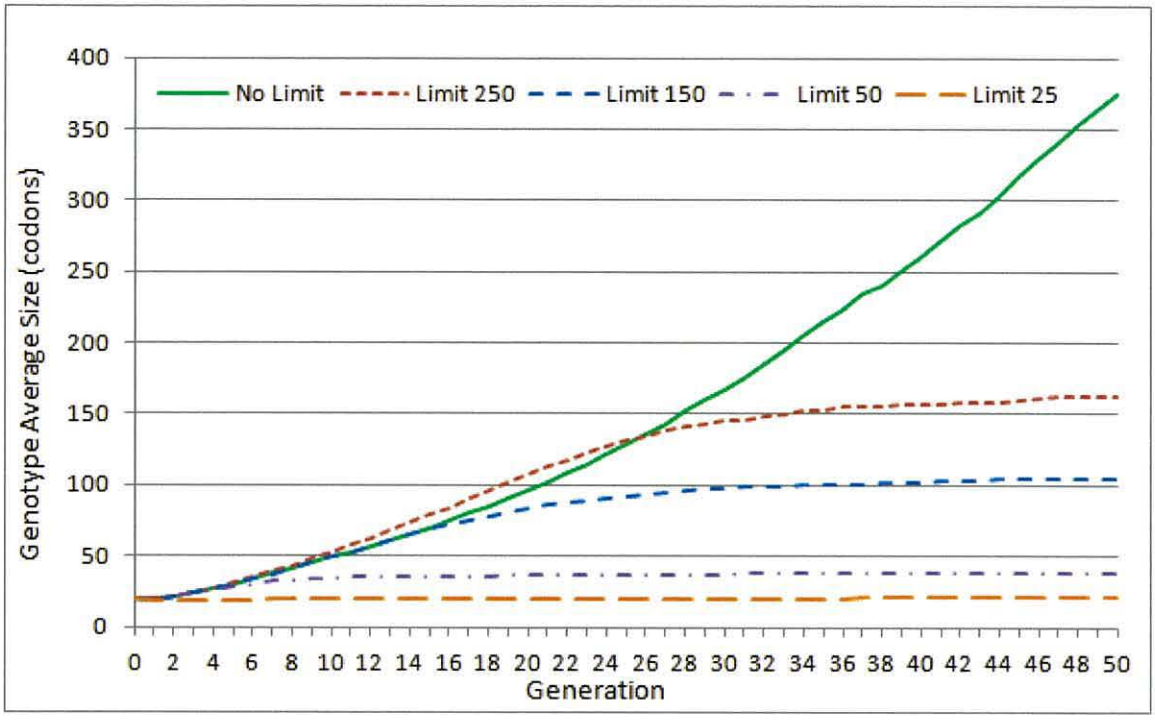


Figure 7.13: Average genotype size in codons of individuals per generation of Grammatical Evolution in the Santa Fe Trail problem with and without genotype size restrictions (25, 50, 150 and 250 codons).

Regarding the impact of the enforcement of the maximum genotype size limit on the genome size of individuals in population level, Figure 7.13 displays the average size (codons) of the genotypes of the individuals per generation during the evolutionary runs of the conducted experiments. The graph shows that the enforcement of the limit resolves the genotype bloat issue and furthermore that it does not result into fixed-length genotypes in a population. Namely, genotypes in the population are still of variable-length during the evolutionary run (the average sizes are smaller than the enforced limits) even though it seems that their average sizes are stabilizing gradually with time as the evolution progresses, especially when this limit has smaller values.

7.8.4 Discussion

This section discusses the unique feature of CGE, their synergy, and their contribution to the performance displayed by the algorithm on the benchmarking problems conducted in this work.

Constituent Genes

Constituent genes state with their addition in the grammar an emergent search bias toward the use of potentially larger and more useful segments of code without restricting the search space defined by the original grammar (existing non-terminals and productions are not replaced or removed). Instead, if the grammar which dictates the form of the constituent genes allows the creation of structures that cannot be constructed by the grammar where they are added, then they will increase the space of possible hypotheses that can be constructed.

In the case of CGE, the original grammar used by the algorithm in this work is the *BNS-BS Blueprint* (Listing 7.7) which states a declarative language bias. Constituent genes are constructed using an unbiased grammar (BNF-Koza artificial and maze versions) therefore, when they are added as productions in the *BNF-BS* grammar, they increase the possible programs that can be constructed by expanding the search space to areas which were excluded. For example, the addition of a constituent gene with phenotype "*move move*" or any conditional statement (which results in nested conditions in the *if* branch of the conditions of the <*behaviour*> non-terminal) will allow the creation of individual phenotypes that otherwise could not be constructed by the *BNF-BS* grammar definition.

Regarding the random process by which the candidate constituent genes are created, the statistical results in Table 7.14, Table 7.15, Table 7.17, and Table 7.18 reveal that even though it seems to be able to find segments of useful code (but mostly of a small size), it requires substantial additional computational effort before each GE run and furthermore it is computational ineffective (only approximately 42% of randomly created genes have a valid phenotype). The use of a more competent creation mechanism (e.g. evolution or knowledge encoding) that results in better and more diverse candidate constituent genes in a computationally more efficient way, would increase the performance of the CGE in terms of success rate, solution quality, and computational effectiveness.

Behaviour-Switching Approach

This feature is inspired by the obvious usefulness and applicability of "condition checking" and "memory" competences in domains where *decisions* and *states* play a significant role such as in agent-oriented decision-related problems. The *BNF-BS* grammar definition enforces the programs to start with a conditional statement and incorporates a kind of memory due to the structure of the nested statements it defines which is of the form shown in Listing 7.4. The aim of the behaviour-switching approach is the encoding and utilisation of knowledge that is generally applicable not only in a single problem instance but in a problem domain. It could be thought as something similar to stating declaratively in a grammar definition the correct syntax of flow control structures of a programming language instead of making them subject of search as well, in order to not increase unnecessarily the search space when a computer program is to be evolved.

In this work, the behaviour-switching approach has been applied successfully, displaying similar or better performance than when the original grammar is used, either in isolation by extending the BNF-Koza grammar and stating a declarative search bias, or in conjunction with constituent genes where it enforces a declarative language bias. The problems it has been applied (artificial ant and maze searching) are similar, therefore application in more diverse and difficult problems is required in order to conclude whether this approach is actually general and can be applied successfully in the domain of other agent-oriented problems.

Synergy of Behaviour-Switching and Constituent Genes

The benchmarking of the unique features of CGE revealed that none of them can achieve in isolation, or at least to approximate, the performance that CGE displayed on all the problems it has been applied in this work. When only constituent genes or only behaviour-switching is used, the defined search space is substantially larger than the search space which is defined by the *BNF-BS* grammar definition after its modification with the addition of the constituent genes just before a CGE evolutionary run.

The *BNF-BS Blueprint* grammar definition of Listing 7.7 states the same language bias with the grammar *CB #2* of Listing 6.11 in section 6.5. Therefore, it is the main contributor in the high success rate of CGE but restricts the search space with the risk of

excluding good solutions for a problem. It is recalled that the *CB #2* grammar achieves an almost absolute success rate (93%) on the Santa Fe Trail problem with the drawback that it could not find a solution requiring less than 405 steps. For example, this grammar cannot construct, due to the declarative language bias it states, the solution in Listing 7.9 which was found by CGE on the Santa Fe Trail and which requires only 337 steps.

When the behaviour-switching approach is used without restricting the original search space, it states the same search bias with the grammar *CB #5* of Listing 6.14 which resulted in a success rate 35% on the Santa Fe Trail (they are actually the same grammars). The constituent genes feature in isolation results in grammar definitions similar to the grammar definitions of section 6.4, which declare a search bias, and especially with the grammar *BB #2* of Listing 6.3 (displaying 30% success rate in Santa Fe Trail) when a genes pool of size 3 is used by CGE. The substantial difference is that the search bias imposed by constituent genes is emergent and not declarative as in *BB #2*.

With the addition of the constituent genes to the language biased grammar *BNF-BS*, the defined search space increases alongside with the possible hypotheses that can be constructed due to the addition of useful building blocks like in the sample grammar of Listing 7.26. Therefore, because of the variety of the added constituent genes during a series of evolutionary runs, the possible hypotheses that can be constructed increase and with it the possibility of finding good solutions (requiring fewer steps) increases as well.

Genotype Max Size Limit

The contribution of this feature in CGE is the prevention of genotype bloating without resulting in fixed-length genotypes in a population and in the economy of the required computational effort. It is shown that it decreases in Grammatical Evolution the volume of the genotype-to-phenotype mappings (number of expansions of a non-terminal to one of its productions) and the execution time they require. The smaller this limit is, the larger is the gain in computational effort. This feature results alongside with the other two features of CGE to smaller genotypes and consequently to less computational effort regarding the mapping process. This computational effort saving could probably counterbalance the additional computational effort required for the creation of the genes pool.

But there is the risk of the selection of a limit value below a problem and grammar dependent threshold that will affect negatively the success rate of evolutionary runs. Therefore, further work regarding this feature is required to fully investigate its impact on Grammatical Evolution and CGE. Some of the questions that could be investigated in future work are as follows. What is the critical threshold that affects performance in Grammatical Evolution and CGE taking into account the grammar and the problem in question? What is the impact on other aspects such as phenotype size and bias of the evolutionary search? What is the relation with wrapping, namely how the max limit affects performance in a problem taking into consideration the effective genome size? How limiting the genome size affects genotypic diversity? Research toward this area could probably result in a theoretical scheme that highlights the trade-off between the impact on finding a solution and the expected gain in computational effort, and which could inform the suggested values this limit should take in specific cases (for different problems and grammars).

7.9 Experimental Results Conclusions

Constituent Grammatical Evolution has been benchmarked against Grammatical Evolution in four problems – Santa Fe Trail, Los Altos Hills, Hampton Court Maze, and Chevening House Maze – and the results show that it improves the later in all of them. Consequently, it seems to be an effective and efficient algorithm for the class of problems where the subject of search is the conditional behaviour of an agent in a given environment.

Due to its unique features, CGE achieved some impressive experimental results. Particularly, it improves the standard Grammatical Evolution algorithm in the Santa Fe Trail problem, whether the later uses the BNF-Koza or the BNF-O'Neill grammar definition. CGE achieves a success rate of 90% against 10% and 78% respectively for GE. These are the averaged values over 5 experiments of 100 runs for each. The best success rate achieved by CGE in an experiment of 100 runs is 94%. Using standard GE with BNF-Koza and BNF-O'Neill, the best success rates achieved were 13% and 81% respectively.

Furthermore, the most efficient solution found by CGE in the Santa Fe Trail problem requires only 337 steps. This solution is much better than then best solutions mentioned by Koza (1992, p.154) and O'Neill and Ryan (2003, p.56) which require 545 and 615

steps, respectively. Instead, the most efficient solutions found experimentally by Grammatical Evolution using BNF-Koza and BNF-O'Neill, require 415 and 607 steps respectively.

Besides the required steps, CGE generally finds solutions of smaller genotype size than Grammatical Evolution. The size of the smallest solution found by CGE is just 73 bits. Instead, the smallest solutions found by GE using BNF-Koza and GE using BNF-O'Neill, have size 142 bits and 90 bits respectively. Regarding the size of the phenotype, GE was better than CGE. The solution with the smallest phenotype found by CGE uses 13 operators and solves the problem in 519 steps. Instead, GE using BNF-Koza and GE using BNF-O'Neill found solutions of 9 operators which solve the problem in 589 and 611 steps respectively.

Another important finding from the conducted experiments is that Grammatical Evolution did not find a solution in the Los Altos Hills problem while CGE was able to find a solution with a 9% success rate. Also, CGE proved to be very effective in the Hampton Court Maze because of its ability to overcome the deceptive local optimum of the problem due to its constituent genes.

Performance measures in the Santa Fe Trail show that CGE requires less time and consequently less processing power than GE using BNF-Koza, for the creation and evaluation of the population of individuals of a generation during an evolutionary run.

Finally, the benchmark of two of the unique features of CGE – constituent genes and behaviour-switching approach – show that, even though both improve Grammatical Evolution, none of them in isolation achieves the high success rates of CGE. It is the synergy of these features which allows the restriction and biasing of the original search space of the problem in question without excluding good solutions, that makes CGE so successful in the problems it has been benchmarked. Regarding the third unique feature of CGE – genotype size maximum limit – it is shown that it decreases significantly the computational effort of the genotype-to-phenotype mappings.

Chapter 8

Conclusions and Future Work

8.1 Discussion

Grammatical Evolution is a flexible and promising grammar-based evolutionary algorithm with unique features inspired from molecular biology such as genetic code degeneracy, due to its genotype-to-phenotype mapping mechanism; and genetic material reuse, due to the genome wrapping it applies. Even though Grammatical Evolution shows competence on a series of problems where it has been applied, the experiments conducted in this study cast doubt about its effectiveness and efficiency on problems where the subject of evolution is the behaviour of an agent, something that was first demonstrated by Robilliard, et al. (2006) in the Santa Fe Trail problem. Problems of this type have some characteristics of real world problems, such as many local optima and large search spaces, making them challenging and difficult for evolutionary algorithms to efficiently solve them (Langdon and Poli, 1998a; Hugosson, Hemberg, Brabazon and O'Neill, 2010).

The experimental results show that Grammatical Evolution does not outperform Genetic Programming in the Santa Fe Trail problem as GE literature claims, and that it is not able to solve the Los Altos Hills – a more difficult problem than the Santa Fe Trail. In addition, it is shown that Grammatical Evolution does not achieve competent results in two maze searching problems: the Hampton Court Maze, and the Chevening House Maze.

Furthermore, Grammatical Evolution when it uses the biased search space mentioned in the GE literature (O'Neill and Ryan 2001; 2003, pp.55-58) is not able to find solutions of similar or better quality than those found when it uses the original search space of the problem in question. It is readily apparent from the results that the Grammatical Evolution feature of defining the search space of a problem with a BNF grammar definition is a subject of implicit bias by the designer with the drawback of the possibility that good solutions of the problem in question may be excluded. Additionally, there are many important issues that still require resolution as has already been reported in the Grammatical Evolution literature.

Constituent Grammatical Evolution (CGE) has been developed during this work as an improvement of Grammatical Evolution which copes with how to apply grammatical bias without excluding good candidate solutions, how to reduce the impact of destructive crossover events through modularity, and how to resolve the genotype bloating issue through the enforcement of a genotype size limit – the last two are known issues of Genetic Programming as well. CGE introduces three unique features – constituent genes, conditional behaviour switching and genotype maximum size limit – and it is shown that it significantly improves Grammatical Evolution on all the problems it has been benchmarked on. Also, it is able to find solutions requiring fewer steps than those found by Grammatical Evolution in the experiments conducted in this study – regardless if the later uses the original or the GE literature biased search space – and those mentioned by Koza (1992, p.154) and O'Neill and Ryan (2001; 2003, p.56).

Particularly, Constituent Grammatical Evolution aims to restrict and shape the search space and reduce at the same time the risk of excluding good solutions for the problem in question. It encodes general problem domain knowledge to state a declarative language bias (behaviour-switching) in order to increase the chance of finding a solution; it applies modularity (constituent genes) to bias the search toward useful areas; and aims to reduce the risk of excluding good solutions by extending the restricted search space with the addition of initially excluded areas where probably good solutions exist.

Additionally, constituent genes state an emergent search bias toward larger and probably more useful constructs and reduce the impact of destructive crossover due to modularity. Namely, by protecting useful building blocks from being disrupted during crossover and reducing the genotype size making disruptions less probable. Furthermore, the addition of the constituent genes in the grammar does not require more non-terminals. CGE eliminates also the genotype bloat phenomenon through the enforcement of a genotype size maximum limit which decreases dramatically the computational effort of the genotype-to-phenotype mappings, retains variable length genomes in the population, and does not affect the performance in terms of success rate and solution quality when an appropriate limit value (above a threshold) is used.

The CGE approach to explore the search space is similar to Whigham's (1995a; 1995b; 1996). He uses declarative language bias to restrict the search space and learnt bias to shape it (search bias) by identifying useful building blocks and encapsulating them as new

productions during the evolutionary run of a grammar-based GP system which does not employ the genotype-to-phenotype mapping process (the grammar is used to dictate legal derivation trees of individuals). The most crucial difference with this approach is that CGE does not aim just to shape the search space through an emergent modularity mechanism, the constituent genes. Instead, CGE aims to extend the search space adding probable useful areas. Because constituent genes are created with the unbiased grammar of the problem in question, their addition in the biased grammar allows, depending on the form of the added constituent genes, the construction of possible hypotheses that could not be constructed before, and furthermore allows during a series of CGE consecutive runs the potential construction of any possible hypotheses of the original unbiased grammar.

The current mechanism for the creation of the constituent genes (which is based on randomness) provides, as has been shown, a limited possibility of including new and useful areas in the initially restricted search space. Therefore, future research is required targeting a more competent genes pool creation mechanism and the development of a framework for the assessment of the coverage and quality of the areas where the search space is expanded.

Furthermore, CGE demonstrates how easily powerful concepts like restriction and shaping of the search space and modularity can be implemented in Grammatical Evolution targeting impressive performance improvement. The research undertaken in this work and the promising results support the belief that Grammatical Evolution can incorporate in a similar easy and flexible way due to its unique features, more concepts inspired from biology and computer science in order to confront issues either intrinsic or inherited from Genetic Programming and to constitute a competent member of the evolutionary algorithms family for solving even problems that today seem to be intractable in the evolutionary computation field.

Finally, the investigation undertaken in this study required the development of a series of tools: The jGE Library; the jGE NetLogo extension; NetLogo models for the simulation of the problems in question; and NetLogo models for the simulation of evolutionary runs with GE and CGE on those problems. The jGE Library was the first published implementation of Grammatical Evolution in the Java programming language. The jGE NetLogo is the only implementation of Grammatical Evolution for the NetLogo modelling environment. The Santa Fe Trail and the Los Altos Hills models in NetLogo are the only

published and widely available simulations (to the author's knowledge) which allow the user to execute and investigate, through a GUI, a specific (designed in priori) ant's control program.

The above tools are freely available under the GNU general public licence from the jGE and NetLogo web sites. It is hoped they will bring together two different communities, namely to allow NetLogo modellers to become familiar with and use Grammatical Evolution within their models, and researchers interested in evolutionary computation to use Grammatical Evolution directly within a multi-agent programmable environment such as NetLogo, for the evolution of the behaviour or morphology of agents.

8.2 Summary and Conclusions

Natural evolution and genetics inspired the development of a new paradigm of computational problem solving, named evolutionary algorithms, which formed the foundation of the field of evolutionary computation. Chapter 2 provided a survey of the field focusing on its history, the main approaches, the current issues, the research findings and the future directions. The survey revealed that even though evolutionary algorithms did not yet model effectively the meta-learning based progress of biological evolution they are promising candidates for tackling problems involving the evolution and emergence of agents' behaviour, morphology, and design. A recent development of the field is Grammatical Evolution, a form of Genetic Programming, which already displays popularity and a variety of applications and variations. Even though, as has been shown in Chapter 2, a number of unsolved issues exist, its ability to create arbitrary structures and valid executable code in any programming language – due to the use of a BNF grammar definition and its unique genotype-to-phenotype mapping mechanism – makes it an appealing approach for problems where the subject is the emergence of a program that controls the behaviour of an agent.

Chapter 3 described the architecture and main components of the Java Grammatical Evolution (jGE) Library which provided the basic toolkit for the experiments conducted in this study. The chapter demonstrated the use of jGE in three proof-of-concept experiments for solving a Hamming distance, a symbolic regression, and a trigonometric identity problem. Also, a comparison of jGE with two other Grammatical Evolution

implementations, libGE and GEVA, was performed and presented. The main difference between jGE and these implementations is that jGE is designed not only to provide an implementation of Grammatical Evolution but to constitute an extendable framework for experimentation in the area of evolutionary computation. jGE's open architecture and extendibility facilitate the implementation of various evolutionary algorithms and genotype-to-phenotype mapping mechanisms, the incorporation of natural-inspired concepts, the interaction with the environment, and the incorporation of agent-oriented principles.

Also, during the development of jGE, two Java issues revealed. The first was the compiling and execution time of the Java code for fitness assignment to an individual. These tasks proved to be extremely time consuming when the Java 2 Standard Edition compiler and interpreter provided by Sun Microsystems Inc. (now by Oracle Corporation) were used. The issue was tackled with using the Dynamic Class Loading and Introspection features of the Java Virtual Machine, and the Jikes compiler provided by IBM Corporation. The second issue was that the Java compiler cannot compile a method with bytecode size larger than 64Kb. This was tackled with the refactoring of the fitness assignment class so that no method exceeds this limit.

Chapter 4 described and discussed three extensions of the jGE Library. The first implements and investigates the role of prior knowledge in evolutionary runs which has been shown in the chapter to improve the effectiveness and efficiency of jGE in symbolic regression and trigonometric identity problems. The second extension of jGE investigates the effect of the natural-inspired concept of population thinking in a trigonometric identity problem revealing that it is a promising approach meriting further investigation and experimentation. The last extension described in Chapter 4 is the jGE NetLogo extension. This extension enables the utilisation of a subset of the features and methods of the jGE Library in NetLogo models. The extensions and the experimental results presented in Chapter 4 demonstrated the easiness of extending jGE and set the targets of future versions of the jGE Library such as incorporation of more natural-inspired principles and concepts (species, families, and shared knowledge to name a few), and experimentation with jGE in NetLogo models.

Chapter 5 confirmed the claim of Robilliard, et al. (2006) that the comparison of Grammatical Evolution and Genetic Programming in the Santa Fe Trail problem, as it is

conducted in the GE literature, is not a fair one because Grammatical Evolution uses a BNF grammar definition (BNF-O'Neill) that defines a search space which is not semantically equivalent with the original as defined by Koza (BNF-Koza). Namely, that it states a declarative language bias which restricts the original search space used by GP in these benchmarks. Also, it has been shown that GE using BNF-O'Neill and a configuration similar to this used in the GE literature (O'Neill and Ryan, 2001; 2003, p.56) is not able to find solutions requiring less than 607 steps. Consequently good solutions requiring fewer steps are excluded due to the restriction of the search space. Additionally, the experimental results showed that GE using BNF-O'Neill does not perform much better than random search, when fixed-length genomes of 500 codons are used, displaying a success rate approximately 67% against 50%. Also, the experimental results revealed that GE using BNF-Koza (the original search space), displays a success rate of approximately 10%, therefore Grammatical Evolution does not outperform Genetic Programming in this benchmark as claimed in the GE literature (O'Neill and Ryan, 2001; 2003, pp.57-58) where Genetic Programming appears to achieve a success rate of approximately 65% and 15%, when the later uses or does not use the solution length constraint, respectively. Furthermore, it has been proved experimentally that GE using BNF-Koza is capable of finding better solutions that those mentioned in Koza (1992, p.154) and O'Neill and Ryan (2001; 2003, p.56). The best solution found in the conducted experiments requires 415 steps when the standard configuration is used and 377 steps when a fixed-size genomes configuration is used instead of variable length as in standard Grammatical Evolution.

Chapter 6 investigated the effects of grammatical bias in the performance of Grammatical Evolution on the Santa Fe Trail problem through a series of experiments where a variety of biased grammars were used. These experiments were stimulated from the results of the previous chapter and demonstrated the effects of grammatical bias on the effectiveness (success rate) and efficiency (solution quality) of Grammatical Evolution when declarative language or search bias is applied. Namely, the results showed that a strong declarative language bias may result in a high success rate but with the risk that this may be achieved at the expense of the solution quality. Instead, promising results were achieved in both success rate and solution quality with the application of a search bias which was applied in the grammar in two ways. First, with the utilisation of modularity using building blocks consisting of useful code segments implemented as additions of

production rules in the grammar, and second with the encoding in the grammar of a generally applicable domain knowledge of the class of agent-oriented problems implemented as checks (conditional statements) in the start of the ant control program. Additionally, the experimental results revealed the important role that the number of the added building blocks plays in performance when modularity is applied in Grammatical Evolution.

The findings of the two previous chapters inspired and directed the development of a variation of Grammatical Evolution, named Constituent Grammatical Evolution (CGE) which was described and benchmarked in Chapter 7. The experimental results showed that CGE achieves a success rate of approximately 90% on the Santa Fe Trail problem with the best solution found requiring only 337 steps. Without restricting the search space through the behaviour switching feature, namely using only constituent genes and the unrestricted original search space, it has been shown that CGE achieves a success rate of 37% improving GE using BNF-Koza (10%) as well. CGE and GE benchmarked in three additional problems (Los Altos Hills, Hampton Court Maze, and Chevening House Maze) and has been shown that the proposed approach improves Grammatical Evolution in all of them as well regardless of whether GE uses the original or the biased search space. Finally, the chapter investigated various aspects of Constituent Grammatical Evolution such as the length of the genotypes and phenotypes of the solutions, the processing requirements, and the effectiveness of each of its unique features in isolation. The results displayed that Constituent Grammatical Evolution produces, in general, solutions of smaller genotype size but of larger phenotypes than Grammatical Evolution. Also, the comparison of CGE with GE using BNF-Koza showed that the former is more efficient in terms of the required processing power per generation of a population during an evolutionary run.

Regarding two of the unique features of CGE, constituent genes and conditional behaviour-switching, it has been shown in Chapter 7 that none of them in isolation achieves in the conducted benchmarks the success rate of CGE, but both of them, especially constituent genes, achieve in general better performance than Grammatical Evolution when the original (unrestricted) search space is used. Also, regarding the third feature, the genotype maximum size limit, it has been shown that it does not affect the success rate of CGE or GE when an appropriate value is used and that it reduces

significantly the total computational effort of the genotype-to-phenotype mapping process during an evolutionary run.

8.3 Review of Aims and Objectives

The first objective of the research undertaken in this thesis was to perform a literature survey on the area of evolutionary computation, and Grammatical Evolution in particular. This revealed that there is not a unique configuration of an evolutionary algorithm which effectively solves any class of problems. Also, achieving a proper balance between exploration (reproductive mechanisms) and exploitation (selection mechanisms) is one of the most important factors in designing and configuring an evolutionary algorithm. Another important aspect is the ability of an evolutionary algorithm to preserve diversity of the population in order the premature convergence of the population to be avoided. The survey revealed also that evolutionary algorithms are a promising approach for tackling complex computational problems such as search, optimization, machine learning, self-adaption, emergence of agent behaviour or morphology, and more. Two standard benchmarking problems of evolutionary algorithms are, artificial ant and maze searching, which appear in a variety of instances: John Muir Trail, Santa Fe Trail, Los Altos Hills, Hampton Court Maze, Chevening House Maze, and more. The survey of the Grammatical Evolution literature showed that this grammar-based algorithm outperforms Genetic Programming in one of these problems, the Santa Fe Trail, but this came later in question.

The second objective was to identify important Grammatical Evolution issues and survey possible resolutions under research. The following issues were highlighted: change of a benchmark definition due to the implicit bias of the search space through the BNF grammar definition and exclusion of good solution of the problem in question; destructive crossovers; genotype bloating; dependency problems; and low locality of genotype-to-phenotype mapping. A variety of Grammatical Evolution variations have been developed over the last years, each of them trying to improve Grammatical Evolution and/or to resolve some of its issues, with the most noticeable of them to be GAuGE, LINKGAUGE, Grammatical Swarm (GS), $(GE)^2$, mGGA, PGE, πGE, GDE, and TAGE. It has been discovered during the survey conducted in this work that none of them (except TAGE which uses a tree-adjunct grammar) is reported in the GE literature to have demonstrated resolution of these issues displaying significant high performance in agent oriented

problems. For example, some of them were applied in the Santa Fe Trail problem using a variety of different setups and parameters showing the following success rates against standard Grammatical Evolution: 12% for TAGE against 3% for GE (Murphy, O'Neill, Galván-López and Brabazon, 2010), 3% for πGE against 13% for GE (O'Neill, Brabazon, Nicolau, Garraghy and Keenan, 2004), 43% for GS against 58% for GE (O'Neill and Brabazon, 2006a), and 17% for GDE against 28% for GE (O'Neill and Brabazon, 2006b).

The investigation of the claim of Robilliard, et al. (2006) was the third objective of the thesis. Therefore, a series of experiments was conducted using standard Grammatical Evolution, a fixed-length genome variation of Grammatical Evolution with and without wrapping, and random search, to discover the effect of the search spaces in question on the performance of Grammatical Evolution and random search. The results confirmed the claim and revealed that the language bias imposed in the original search space is very strong and excludes good solutions to the problem.

The experimental results of the previous objective answered also the next objective of the thesis which was to benchmark the performance of Grammatical Evolution on the Santa Fe Trail when the original search space is used and find out whether it still outperforms Genetic Programming. The results showed that Grammatical Evolution using the original search space and a configuration similar to that mentioned in the GE literature does not outperform Genetic Programming and furthermore that when the biased search space is used it does not perform much better than random search.

The last objective of the thesis was to tackle some of the most noticeable issues of Grammatical Evolution with the hope to improve its effectiveness and efficiency in benchmark problems where the subject of evolution is the behaviour of an agent in a given environment. The proposed variation of Grammatical Evolution, named Constituent Grammatical Evolution, aims to tackle and/or reduce the impact of three Grammatical Evolution issues – implicit bias of the original search space during design and exclusion of good solutions; destructive crossover events; and genotype bloating – introducing three features: conditional behaviour-switching, constituent genes concept, and genotype size limitation. It has been shown that CGE improves Grammatical Evolution in all problems it was benchmarked.

Finally, when the research of this thesis started, there was no publicly available implementation of Grammatical Evolution in the Java programming language that would enable benchmarking and experimentation with this algorithm in any Java enabled platform in order to take advantage of some of the features of this language such as pure object-oriented design, high-level data constructs, and platform independent code. For this reason the jGE Library has been developed, which is intended to provide an open and extendable framework for experimentation with evolutionary algorithms, beyond Grammatical Evolution.

Also, the benchmarking of Grammatical Evolution and its proposed improvement required the development of simulations of the problems in question: Santa Fe Trail, Los Altos Hills, Hampton Court Maze, and Chevening House Maze. These were implemented in the NetLogo modelling environment. Furthermore, an extension of jGE for the NetLogo modelling environment was developed as well as a series of NetLogo models that simulate Grammatical Evolution and Constituent Grammatical Evolution evolutionary runs in the benchmark problems conducted in this work.

8.4 Summary of Contributions

A summary of the contributions of this work is listed below.

- the development of the jGE Library;
- the development of the jGE NetLogo extension;
- the development of NetLogo models simulating instances of the artificial ant problem - Santa Fe Trail and Los Altos Hills – which allow the design and benchmark of candidate solutions using a GUI;
- experimental evidence to support the claim of Robilliard, et al. (2006) regarding the restricted search space being used in the Grammatical Evolution literature;
- experimental evidence that shows that Grammatical Evolution does not outperform Genetic Programming in the Santa Fe Trail;
- the demonstration of the effects of declarative grammatical bias through building blocks and knowledge encoding in the performance of Grammatical Evolution on the Santa Fe Trail problem;

- experimental evidence highlighting the importance of the number of the added building blocks in the grammar when Grammatical Evolution is utilising modularity;
- the development of a new algorithm, Constituent Grammatical Evolution (CGE), which utilises modularity and grammatical bias to significantly improve Grammatical Evolution on agent-oriented problems in terms of success rate and solution quality (benchmarked on Santa Fe Trail, Los Altos Hills, Hampton Court Maze, and Chevening House Maze);
- the benchmarking of Grammatical Evolution on artificial ant and maze searching problems, highlighting the effects of language bias on its effectiveness and efficiency; and
- demonstration of the impact of genotype bloat on the computational effort of the genotype-to-phenotype mapping process in Grammatical evolution and the dramatic decrease of this effort, without affecting the performance in terms of finding a solution, when a genotype size limit above a problem specific threshold is enforced.

Finally, the work of this thesis has resulted to the publication of five papers (see Appendix A) which demonstrated the progress of the work and which comprised the milestones of this study. Also, the jGE Library is publicly available on its web site (aiia.bangor.ac.uk/jge) and it is still under continuous development at the School of Computer Science of Bangor University (Artificial Intelligence and Intelligent Agents Research Group). Researchers and students from around the world have shown an interest in the jGE Library (see Appendix D). The interest in jGE is also indicated by the links to the jGE web site which can be found in the official site of GE (www.grammatical-evolution.org, section: other implementations), in the wikipage of the Grammatical Evolution algorithm (en.wikipedia.org/wiki/Grammatical_evolution), in the wikipage of Genetic Programming (en.wikipedia.org/wiki/Genetic_programming), and in the EJC Project web site (www.cs.gmu.edu/~eclab/projects/ecj).

8.5 Future Work

The results and conclusions of the investigation undertaken during this work as well as the promising benchmarking results of the Constituent Grammatical Evolution algorithm raise

new questions requiring answers and indicate directions of possible further research. The future directions of study that are suggested are as follows: development of additions and extensions for the jGE Library; an exhaustive study of Grammatical Evolution performance in agent-oriented problems; thorough investigation of Constituent Grammatical Evolution, benchmarking in more problems, and research for possible further improvements; evolution of complex agents that utilise shared knowledge; and investigation of possible applications of the concepts of constituent genes and conditional behaviour-switching in, other than Grammatical Evolution, evolutionary algorithms.

8.5.1 jGE Extensions

The jGE Library needs to be extended in order to fulfil the remaining goals of the jGE Project of Bangor University (School of Computer Science - Artificial Intelligence and Intelligent Agents Research Group) as stated in Chapter 3, namely to constitute a general framework for experimentation with evolutionary algorithms, to provide the basis for the creation of an agent-oriented evolutionary system, and to bootstrap further research on the application of natural and molecular biology principles. Consequently, the development and incorporation of more evolutionary algorithms, genetic operators, and evolutionary mechanisms (beyond these described in Chapter 3 and in Chapter 4) is necessary as well as the development of classes for representing environment features, relations between individuals, groups of individuals, and individual's knowledge. Also, it would be interesting to implement in jGE to find out if the application of natural-inspired principles such as population thinking, microevolution, macroevolution, elimination pressure, common ancestor, species, and more, could result in performance improvement of evolutionary algorithms on agent-oriented problems where the problem in question is to find adequate agent behaviours.

Furthermore, other suggested improvements and extensions for the jGE Library are as follows: jGE performance improvement utilising distributed parallel processing; experiments with data logging in a relational database instead of text files; development of experimental data statistics module for gathering information about population convergence and diversity in both genotype and phenotype level, and other quality and quantitative metrics; implementation of more benchmarking problems; development of a

GUI; and implementation of more jGE features and methods in the jGE NetLogo extension.

8.5.2 Grammatical Evolution Benchmarking

Chapter 5 showed that Grammatical Evolution does not outperform Genetic Programming on the Santa Fe Trail when the original search space and a similar to the standard GE literature configuration are used. Indeed, the GE literature review in Chapter 2 revealed that there is no detailed study of the performance of Grammatical Evolution in agent-oriented problems. Therefore, a thorough study and benchmarking is required of the application of Grammatical Evolution and its variations – Constituent Grammatical Evolution, TAGE, $(GE)^2$, PGE, πGE, GAuGE, and Chorus to name a few – using a variety of GE configurations and search engines, in problems where the subject of evolution is the behaviour of agents in static or dynamic environments. Also, the work of this thesis concentrated on the tackling of only three of the Grammatical Evolution issues (see Chapter 7) but there are more issues to be further investigated and confronted such as dependency problems and low locality of the genotype-to-phenotype mapping.

8.5.3 CGE Further Investigation and Improvement

Constituent Grammatical Evolution is a new evolutionary algorithm that improves Grammatical Evolution in the problems it benchmarked in Chapter 7. Consequently, there are still many aspects to be investigated as well as possibilities for further improvements. Namely, different configurations of the CGE parameters need to be tested with different BNF grammar definitions in the same and other benchmark problems. For example, the impact of mutation and crossover rates in CGE's performance using different BNF grammars, how the increase or decrease of the maximum genotype size limit or of the constituent genes pool size affects the results, what are the suggested values of these parameters for specific grammars and problems, and how well CGE performs on other agent-oriented problems of a static or dynamic nature.

Regarding the application to dynamic agent-oriented problems, the current genes pool creation mechanism of CGE (performed once before an evolutionary run) cannot address them effectively. In such problems, the genes pool needs to be dynamic, namely to change during the evolutionary run in order to adapt to the new challenges imposed by the

problem in question. In such a case, there is the issue mentioned by Swafford, O'Neill and Nicolau (2011) of disrupting the fitness of the existing individuals due to the change of the grammar. Further investigation is required toward this area in order to allow the change of the grammar in CGE during the evolutionary run without causing disturbance in fitness. Research on this area using Grammatical Evolution has already been carried out by Swafford, et al. (2011) with promising results using a new operator called *repair* that modifies the genotype of the individual in order that the productions picked during the mapping with the new grammar are the same as the productions picked before the grammar was changed.

Furthermore, research is required toward the development of a sophisticated constituent genes creation mechanism instead of using random search. The utilisation for example of an evolutionary algorithm is expected to produce more competent constituent genes (useful building blocks). This could also be extended with the application of *mosaic evolution* (Mayr 2002, p.243), namely evolving both constituent genes and individuals at the same time but with an uneven rate. Also, alternative evaluation and selection mechanisms of the constituent genes need to be examined than those already used and described in Chapter 7. It is expected that improving these, and probably other, mechanisms of CGE will result in performance improvement of the algorithm in the problems it has been benchmarked. Additionally, the development of a framework for the assessment of the coverage and quality of the areas where the search space is expanded when constituent genes are utilised to expand a restricted search space will facilitate the creation and selection of more competent and useful constituent genes.

Finally, it would be interesting to investigate the performance of GE and CGE in problems requiring larger grammars than these already used in the literature. For example, to try to evolve whole executable programs for a target real programming language rather than having to use an intermediate representation as GE and GP do. An important question is how far GE and CGE can go if a much larger grammar is used.

8.5.4 Utilisation of Shared Knowledge

Experimental results of Chapter 4 showed as expected that prior knowledge increases the performance of Grammatical Evolution. Therefore, a candidate direction of investigation is the utilisation of knowledge, represented with conceptual spaces (Gärdenfors, 2004), as

a fitness value factor. Because it is a fact today that there are no heritable acquired characteristics (Mayr 2002, p.100), it could be assumed that the shared knowledge is a factor which affects the natural selection process and allows existing or new genetic and phenotypic characteristics to possess different importance and value (fitness). Namely, that the evolving shared knowledge will favour different phenotypes as happens in nature with the changes in the environment.

The population thinking principle implementation of Chapter 4 could be extended in the family based approach suggested by Teahan, Al-dmour and Tuff (2005). This can be achieved by incorporating knowledge sharing and consequently simulating in this way social phenomena of living organisms and especially humans (e.g. families). The suggested investigation is hoped to show in which degree phenotypic variation and evolution/sharing of knowledge could lead to better and faster solutions in the area of evolutionary computation.

8.5.5 Toward a new class of Evolutionary Algorithms

The performance increase that is achieved by Constituent Grammatical Evolution, as shown in Chapter 7, due to the incorporation of the concepts of constituent genes, conditional behaviour-switching and genotype size maximum limit, raises the question whether these concepts, especially constituent genes, can be utilised by evolutionary algorithms other than Grammatical Evolution or grammar-based approaches. Therefore, an investigation is required of how these concepts can be applied to other evolutionary algorithms, such as Genetic Algorithm, Genetic Programming, and Parallel Evolutionary Algorithms, and what will be the effect on their performance.

If the application of these concepts to more evolutionary algorithms is proved to be promising, then they should be modelled in a more general way in order to fit the general model of an evolutionary algorithm and be easily applicable to any of them. If such research leads to successful results, it is hoped that a new approach in evolutionary computation could be introduced with the creation of a new class of evolutionary algorithm which incorporates the constituent genes concepts and/or the conditional behaviour-switching concept for agent-oriented problems where the subject of evolution is the behaviour, the morphology, or the design of an agent in a static or dynamic environment.

Appendices

Appendix A Publications

Print Publications

This study has lead to the publications listed below. All of them form a substantial part of the indicated chapters with the major contribution being from the primary author in each case.

Chapter 3 Georgiou, L. and Teahan, W. J. (2006a) "jGE – A Java implementation of Grammatical Evolution". 10th WSEAS International Conference on Systems, Athens, Greece, July 10-15, 2006.

Chapter 4 Georgiou, L. and Teahan, W. J. (2006b) "Implication of Prior Knowledge and Population Thinking in Grammatical Evolution: Toward a Knowledge Sharing Architecture". WSEAS Transactions on Systems 5 (10), 2338-2345.

Chapter 3 Georgiou, L. and Teahan, W. J. (2008) "Experiments with Grammatical Evolution in Java". Knowledge-Driven Computing: Knowledge Engineering and Intelligent Computations, Studies in Computational Intelligence (vol. 102), 45-62. Berlin, Germany: Springer Berlin / Heidelberg.

Chapter 5 Georgiou, L. and Teahan, W. J. (2010) "Grammatical Evolution and the Santa Fe Trail Problem". In Proceedings of the International Conference on Evolutionary Computation (ICEC 2010), October 24-26, 2010, Valencia, Spain, 10-19.

Chapter 7 Georgiou, L. and Teahan, W. J. (2011) "Constituent Grammatical Evolution". In Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI 2011), July 16-22, 2011, Barcelona, Catalonia (Spain), 1261-1268.

On-line Publications

This study has lead to the on-line publications listed below.

- jGE Library Web Site
  aiia.bangor.ac.uk/jge
- jGE Library Wiki Page
  en.wikipedia.org/wiki/Java_Grammatical_Evolution
- Constituent Grammatical Evolution Web Site
  aiia.bangor.ac.uk/cge

On-line References

This study has lead to the development of tools and models which are referenced from a variety of relevant web sites. Below, some of the most important and popular are listed along with their corresponding sections and what exactly are referenced in each of them.

- NetLogo web site, ccl.northwestern.edu/netlogo
  - User Community Models (Santa Fe Ant Trail model)
  - NetLogo Community Extensions for NetLogo 4.1 (jGE NetLogo extension)
- Grammatical Evolution web site, www.grammatical-evolution.org
  - People Working on GE
  - Other implementations (jGE Library)
- GE wiki page, en.wikipedia.org/wiki/Grammatical_evolution
  - Resources (jGE Library)
- GP wiki page, en.wikipedia.org/wiki/Genetic_programming
  - Implementations (jGE Library)
- ECJ web site, cs.gmu.edu/~eclab/projects/ecj
  - Alternative GP Representations (jGE Library)

Appendix B jGE Web Site

Web Site Address (URL): aiia.bangor.ac.uk/jge

The web site presents the jGE Library and provides useful information as well as links for downloading the binary and source code of the jGE Library, the jGE NetLogo extension, and the Santa Fe Trail NetLogo models.

The web site has the following sections:

- jGE Home Page
- Experiments
- Documentation
- Download
- Resources

Regarding the jGE NetLogo extension, the downloadable archive file includes:

1. The binary file.
2. The Documentation and Manual of the extension.
3. The source code of the extension.
4. A Demonstration model of how to use the jGE NetLogo extension.
5. The jGE Library binary archive.
6. Examples of BNF grammars.

The jGE web site is under continuous update and it is expected to be improved in the future with more sections, experimental results, and downloadable material. It is hoped that this web site will make the jGE Library available to a greater audience and that researchers from around the world will show an interest in using it and/or contribute to its improvement and extensions.

Appendix C CGE Web Site

Web Site Address (URL): aiia.bangor.ac.uk/cge

The Constituent Grammatical Evolution (CGE) web site presents the proposed Grammatical Evolution variation and contains links of its implementation in artificial ant and maze searching problems written in the NetLogo programming language.

The web site contains the following sections:

- CGE Home Page
- Algorithm Description
- Benchmarking Results
- Statistics in the Santa Fe Trail problem
- Downloads
- Resources

This web site is still under development and it is expected to be updated and improved with more sections, experiments, findings, and downloadable material based on current and future research.

It is hoped that, the up-to-date publication on the web site of future research findings on Constituent Grammatical Evolution will help the distribution of knowledge and the provisioning of feedback from other researchers showing an interest in the modelling and evolution of agents' behaviour and or morphology.

Appendix D People Interested in jGE

This table lists the people who have shown interest with the jGE Library and who have contacted the author to request more information and/or usage guidelines:

Name	School	Date	Email
Michael Phelan	University College Dublin	10/01/2007	michael.phelan@ucd.ie
Yow Tzu	University of York	14/08/2007	yowtzu@cs.york.ac.uk
David R White	University of York	12/09/2007	drw@cs.york.ac.uk
Marta Stanska	University of Wroclaw	18/11/2007	martastanska@gmail.com
Konrad Drukala	University of Wroclaw	18/11/2007	heglion@gmail.com
Ron King	---	24/11/2007	roncking@cox.net
Elkin Urrea	Graduate Center of NY	10/04/2008	eurrea@gc.cuny.edu
Bruno Fagundes Saldanha	Federal University of Minas Gerais	07/10/2008	brunofs@dcc.ufmg.br
Matt Hyde	Nottingham University	25/11/2009	mvh@cs.nott.ac.uk
Gerard Vreeswijk	Utrecht University (Netherlands)	14/05/2010	gv@cs.uu.nl
Fabian Andres Giraldo	National University of Colombia	09/08/2010	fagiraldo@unal.edu.co
Marc van Zee	Utrecht University (Netherlands)	12/09/2010	m.vanzee@students.uu.nl

Appendix E jGE Library Quick Start Guide

Technical Requirements

In order to use jGE v1.0, you will need Java 5 or later.

Examples

In the source code, you will find a lot of usage examples. Alongside with the jGE API documentation it will help you to start using jGE. Both of them are freely available from the jGE web site (aiia.bangor.ac.uk/jge).

Namely, the source file `bangor/aiia/jge/core/EAExperiments.java` contains some examples of using the jGE classes. Indeed, the source folder `bangor/aiia/jge/junit` contains all the JUnit (JUnit.org, 2006) tests of jGE. These JUnit TestCases can serve as a rich resource of how to use the various classes of the library.

Just keep in mind that anywhere in the source code you see methods like "`ConfigurationSettings.getInstance().get`**xxx**", replace them with a String containing the actual path of the corresponding file or folder in your system. For your convenience, it is suggested that you change the settings of the class `bangor.aiia.jge.util.ConfigurationSettings` to the default values of your system and recompile the class.

Sample Code using Standard GA in a Hamming Distance Problem

```
// The binary string we are looking to find (the solution)
String target = "11100011100010101010101010101010";

// The Problem Specification (implementation of Evaluator)
HammingDistance hd = new HammingDistance(target);

// Instantiation of the Evolutionary Algorithm (SGA)
StandardGA ga = new StandardGA(50, 1, 30, 30, hd);

// Configure the Standard Genetic Algorithm
ga.setFixedSizeGenome(true);
ga.setCrossoverRate(0.9);
ga.setMutationRate(0.01);
ga.setDuplicationRate(0.01);
ga.setPruningRate(0.01);
ga.setMaxGenerations(100);
```

```
// Execute the Evolutionary Algorithm (SGA)
// and get the solution
Individual<String, String> solution = ga.run();
```

Sample Code using Grammatical Evolution in a Hamming Distance Problem

```
// The binary string we are looking to find (the solution)
String target = "11100011100010101010101010101010";

// The Problem Specification (implementation of Evaluator)
HammingDistance hd = new HammingDistance(target);

// Instantiate the BNF Grammar (use for this example
// the file BinaryGrammarFixedLength.bnf)
BNFGrammar bnf = null;
try {
       bnf = new BNFGrammar(
                  new File("[THE_PATH_OF_THE_BNF_GRAMMAR_FILE]")
            );
}
catch(IOException ioe) {
       System.out.println("IOException: " + ioe.getMessage());
}

// Instantiation of the Evolutionary Algorithm (SGA)
GrammaticalEvolution ge = new GrammaticalEvolution(
                              bnf, hd, 50, 8, 20, 40
                              );

// Configure the Grammatical Evolution Algorithm
ge.setCrossoverRate(0.9);
ge.setMutationRate(0.01);
ge.setDuplicationRate(0.01);
ge.setPruningRate(0.01);
ge.setGenerationGap(0.9);
ge.setMaxGenerations(100);

// Execute the Evolutionary Algorithm (GE) and get the solution
Individual<String, String> solution = ge.run();
```

Evaluator

In order to use jGE for ad-hoc problems, you have just to implement a class which will evaluate the individuals of the population (assign a fitness value). This evaluator must implement the `bangor.aiia.jge.core.Evaluator` interface and actually will "encapsulate" the Problem Specification.

Such "out of the box" classes (available in jGE) are the following:

- `bangor.aiia.jge.ps.HammingDistance`

- `bangor.aiia.jge.ps.SymbolicRegression`

For other classes of problems, just create your own implementation of the Evaluator interface and use it with the evolutionary algorithm of your choice (Standard GA, Steady-State GA, or GE).

Mapper

Regarding the mapping, in case of Standard Genetic Algorithms there is no distinction between genotype and phenotype. Consequently, if you do not set a specific implementation of `bangor.aiia.jge.core.Mapper` interface (you can create your own) the default "no-mapping" implementation will be used (`bangor.aiia.jge.core.DefaultMapper`). Then, you have to "decode" the semantics of the binary string of the genotype in your implementation of the Evaluator.

Grammatical Evolution

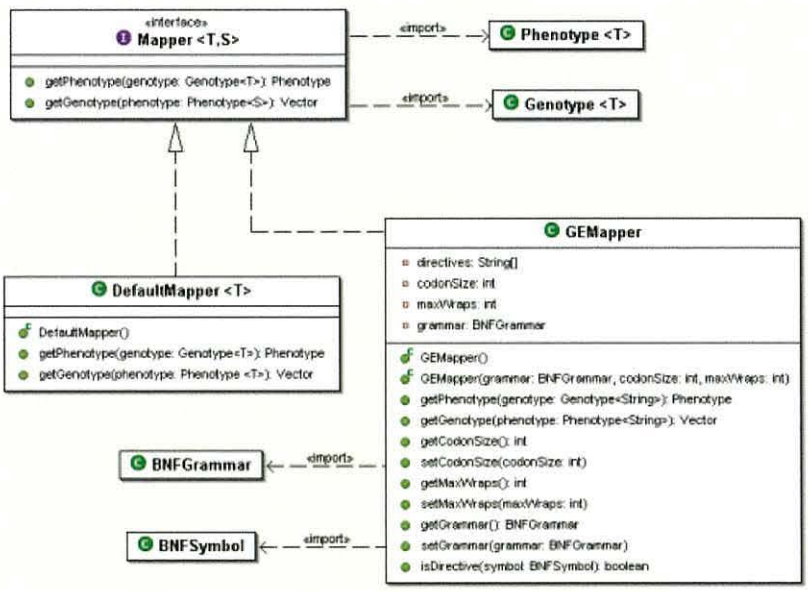
In order to use Grammatical Evolution with jGE in your project, you have to create an instance of the `bangor.aiia.jge.core.GrammaticalEvolution` class and pass to it both the file with the BNF Grammar you will use and your own implementation of the `bangor.aiia.jge.core.Evaluator` interface (which will evaluate the fitness of an Individual of the Population).

Namely, you have to implement a BNF Grammar (which dictates the valid phenotypes of the individuals) and the Evaluator (which assigns a fitness value to each individual).

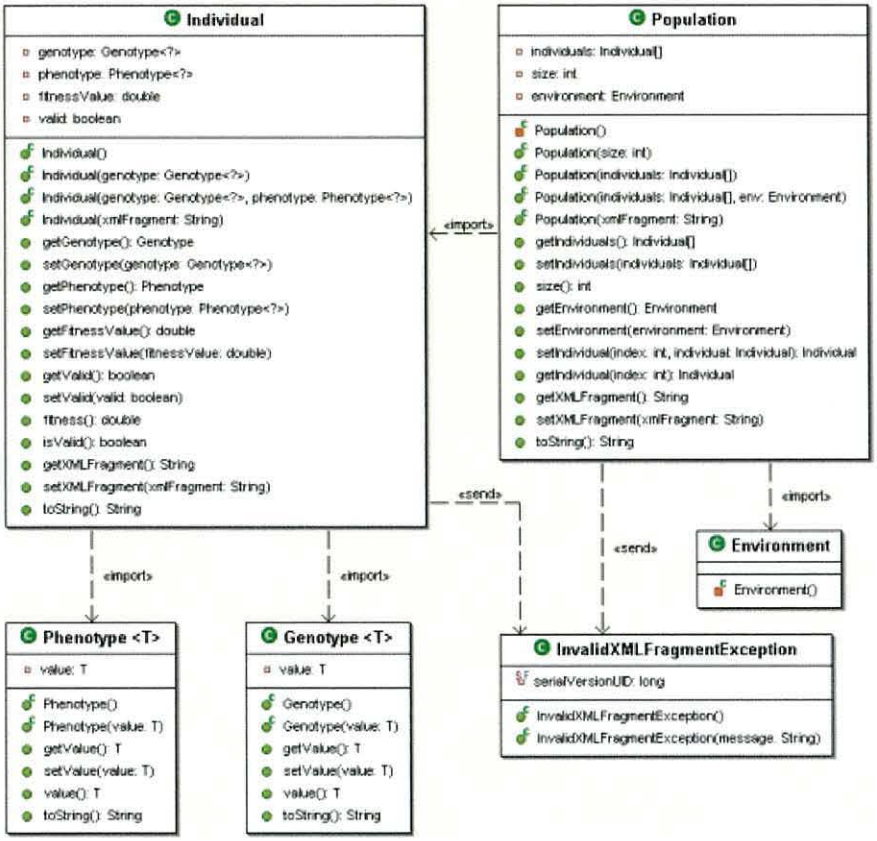
In the accompanying CD, you will find some sample BNF Grammars. You can use them to play around with jGE until you become more familiar and start to create your own.

Appendix F jGE Library Core Class Diagrams

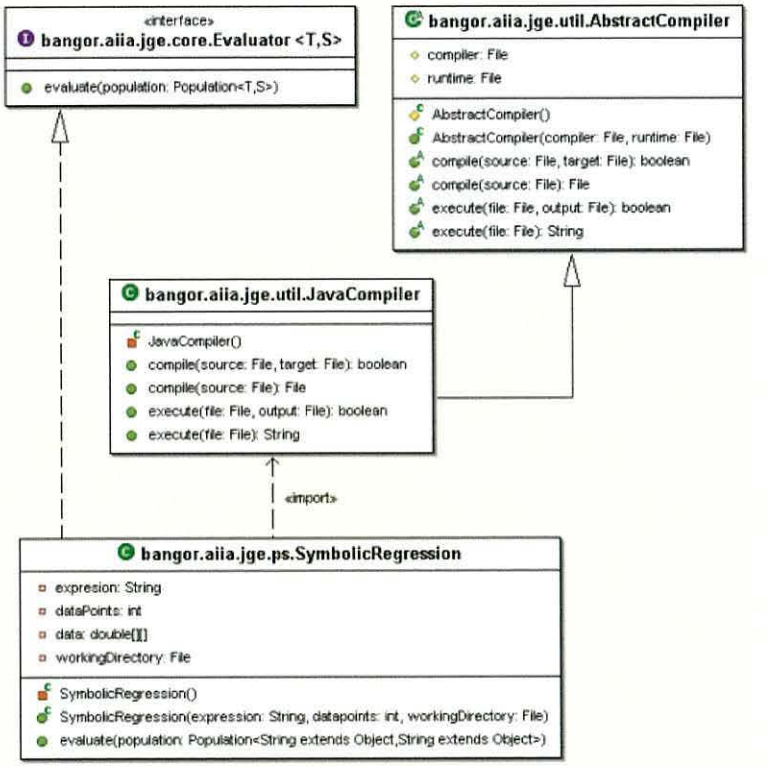
Class diagram of the jGE Mapper component of the bangor.aiia.jge.core package.



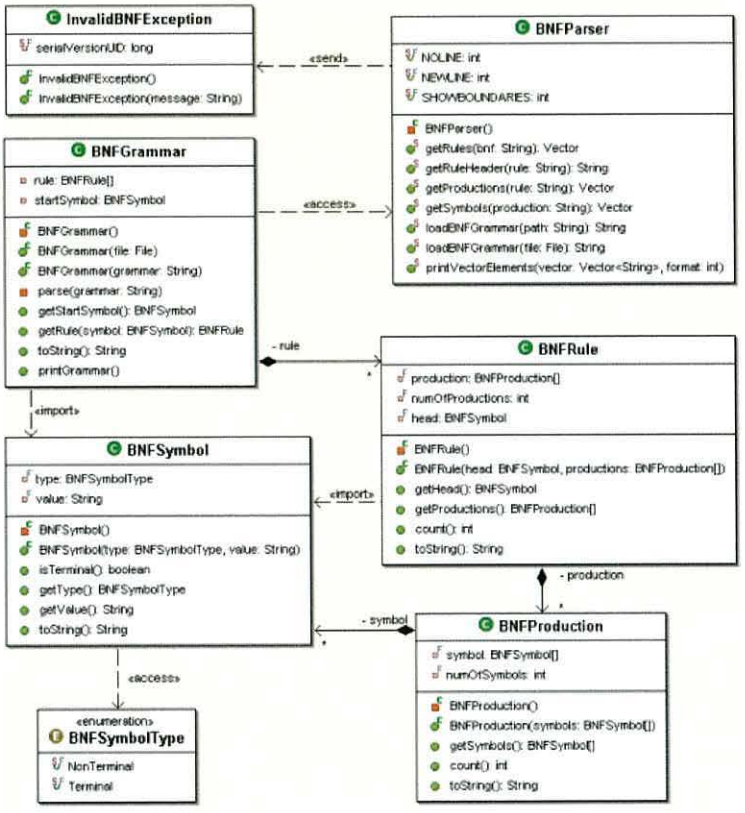
Class diagram of the jGE bangor.aiia.jge.population package.



Class diagram of the jGE bangor.aiia.jge.ps package.



Class diagram of the jGE bangor.aiia.jge.bnf package.



Appendix G jGE NetLogo Extension Procedures

The main procedures (commands and reporters) provided by the jGE NetLogo extension are listed below. The complete documentation can be downloaded from Georgiou (2006) or can be found in the accompanying CD.

jge:load-bnf ***path reference***

This command loads, parses, and keeps in-memory (for later use in the model by mentioning the *reference* value) the BNF grammar definition which is stored in a file with the given path. The argument *path* is a string with the value of the absolute path of the BNF file (or the relative path to the model which calls this command). Indeed, it can be an absolute URL which is necessary in case the NetLogo model runs as Java Applet inside a web browser. The argument *reference* is a string with the name that will be used in the model for referencing the loaded BNF grammar definition (in that way it is possible to load and use more than one BNF grammar definitions in the same model).

jge:bnf-grammar ***reference***

It reports the BNF grammar which is referenced by *reference*.

jge:phenotype ***genotype grammar codonsize maxwraps***

This report executes the genotype-to-phenotype Grammatical Evolution mapping algorithm and reports a string with the corresponding phenotype to the given binary *genotype*. The mapping process uses the BNF grammar definition referenced by *grammar*, size for each codon by the given *codonsize*, and maximum number of genotype wraps by *maxwraps*. If the resultant phenotype is invalid (contains non-terminal symbols) then an empty string is returned.

jge:crossover ***parentA parentB probability***

It reports a NetLogo *List* with the results (two binary strings of variable length) of the application of the standard one-point crossover operation with probability *probability* to the variable-length binary strings *parentA* and *parentB*. Note that the crossover point can be different for each parent resulting in offspring of different genotype sizes.

jge:crossover-fixed-length ***parentA parentB probability***

This primitive reports a NetLogo *List* with the results (two binary strings of the same length) of the application of the standard one-point crossover operation with probability *probability* to the fixed-length binary strings *parentA* and *parentB*. Note that the crossover point is the same for both parents resulting in offspring of the same genotype size with the parents.

jge:mutation ***offspring probability***

It reports a string with the result of the application of the point mutation operation with probability *probability* in the binary string *offspring*.

ge:individual ***codonsize min max***

It reports a randomly generated genotype (binary string) of size between the specified limits (size of codon in bits, minimum length in codons, and maximum length in codons). The binary string will have a random number of codons (each one of *codonsize* bits) in the range between *min* and *max*.

jge:population ***size codonsize min max***

This primitive reports a NetLogo *List* of length *size* with randomly generated genotypes (binary strings). Each genotype (binary string) contains a random number of codons (each one of *codonsize* bits) in the range between *min* and *max*.

Acronyms

Acronym	Full Phrase
ADF	Automatically Defined Function
BNF	Backus-Naur Form
CFG	Context-Free Grammar
CGE	Constituent Grammatical Evolution
CHM	Chevening House Maze
EA	Evolutionary Algorithm
EAP	Evolutionary Automatic Programming
EC	Evolutionary Computation
EP	Evolutionary Programming
ES	Evolution Strategies
FSA	Finite-State Automaton
GA	Genetic Algorithm
GE	Grammatical Evolution
GP	Genetic Programming
HCM	Hampton Court Maze
JVM	Java Virtual Machine
LAH	Los Altos Hills
OOP	Object-Oriented Programming
PEA	Parallel Evolutionary Algorithm
PGA	Parallel Genetic Algorithm
SGA	Standard Genetic Algorithm (Generational)
SFT	Santa Fe Trail
SSGA	Steady State Genetic Algorithm

Glossary

Adaptation An internal change in a system that mirrors an external event in the system's environment (Flake 1998, p.443).

Allele Alternative expression of one and the same gene. For instance, a gene for eye colour has the alleles "brown," "blue," "black," etc. (Pfeifer and Scheier 2001, p.645).

Artificial Life The study of life processes within the confines of a computer (Flake 1998, p.444).

Backus-Naur Form A formalism to write grammars. There are four components to a BNF grammar: A set of terminal symbols, a set of non-terminal symbols, a start symbol, and a set of rewrite rules (Russell and Norvig 2003, p.984).

Baldwin Effect The selection of genes that strengthen the genetic basis of a variant of the phenotype (Mayr 2002, p.311).

Behaviour Control Set of mechanisms that determine the behaviour in which an agent will engage (Pfeifer and Scheier 2001, p.646).

Behaviour What an autonomous agent is observed doing. Always the result of an interaction of an agent with its environment (Pfeifer and Scheier 2001, p.646).

Chomsky Hierarchy Four classes of language (or computing machines) that have increasingly complexity: regular (finite-state automata), context-free (push-down automata), context-sensitive (linear bounded automata), and recursive (Turing machines) (Flake 1998, p.446).

Chromosome A structure contained in every cell of the organism that holds strings of DNA, a macromolecule that serves as a "blueprint" for the build-up and functioning of an organism. A chromosome can be conceptually divided into genes (Pfeifer and Scheier 2001, p.647).

Codon A triplet of bases (or nucleotides) in the DNA coding for one amino acid. The relation between codons and amino acids is given by the genetic code (Ridley 2004, p.683).

Coevolution Two or more entities experience evolution in response to one another. Due to feedback mechanisms, this often results in a biological arms race (Flake 1998, p.446).

Context-Free Grammar In context-free grammars, the left-hand side consists of a single non-terminal symbol. Thus, each rule licenses rewriting the non-terminal as the right-hand side in any context. (Russell and Norvig 2003, p.793).

Context-Sensitive Grammar Context-sensitive grammars are restricted only in that the right-hand side must contain at least as many symbols as the left-hand side (Russell and Norvig 2003, p.793).

Convergence For searches (e.g. genetic algorithms), finding a location that cannot be improved upon (Flake 1998, p.447).

Crossover (Biology) The process during meiosis in which the chromosomes of a diploid pair exchange genetic material. It is visible in the light of microscope. At a genetic level, it produces recombination (Ridley 2004, p.683).

Crossover (Genetic Algorithm) Holland's is the most straight forward form of Crossover: Crossover is applied to two chromosomes (parents) and creates two new chromosomes (offspring) by selecting a random position along the coding and splicing the section that appears before the selected position in the first string with the section that appears after the selected position in the second string, and vice versa (Fogel 2006, p.77).

Darwinism Darwin's theory, that species originated by evolution from other species and that evolution is mainly driven by natural selection. Differs from neo-Darwinism mainly in that Darwin did not know about Mendelian inheritance (Ridley 2004, p.683).

Dynamical System A system that changes over time according to a set of fixed rules that determine how one state of the system moves to another state (Flake 1998, p.449).

Evolution According to Evolutionary Synthesis, evolution is change in the properties of populations of organisms over time (Mayr 2002, p.9). Darwin defined it as "descent with modification." It is the change in a lineage of populations between generations (Ridley 2004, p.684).

Evolution Strategies A species of evolutionary algorithms. The focus of the evolution strategy paradigm was on real-valued function optimisation. Hence, individuals were naturally represented as vectors of real numbers (De Jong 2006, p.25).

Evolutionary Algorithm An umbrella term that includes various types of algorithms that are, in one way or another, inspired by natural evolution. It includes genetic algorithms, evolution strategies, and evolutionary programming (Pfeifer and Scheier 2001, p.649).

Evolutionary Computation The use of evolutionary systems as computational processes for solving complex problems (De Jong 2006, p.1). The field emerged during the 1990s from the adoption of a unified view of the evolutionary problem solvers (De Jong 2006, p.29).

Evolutionary Programming A species of evolutionary algorithms. The evolutionary programming paradigm concentrated on models involving a fixed-size population of N parents, each of which produced a single offspring. The individuals being evolved were finite state machines (De Jong 2006, p.25).

Finite-State Automaton The simplest computing device. Although it is not nearly powerful enough to perform universal computation, it can recognise regular expressions. FSAs are defined by a state transition table that specifies how the FSA moves from one state to another when presented with a particular input. FSAs can be drawn as graphs (Flake 1998, p.451).

Fitness A measure of an object's ability to reproduce viable offspring (Flake 1998, p.451). In biology: (a) The probability that the organism will live to reproduce (viability); (b) a function of the number of offspring the organism has (fertility). In

artificial evolution: The value of a fitness function for a particular individual (Pfeifer and Scheier 2001, p.650).

Fitness Function In artificial evolution, a function that evaluates the performance of a phenotype. Used as an optimisation criterion. Individuals with high fitness have a high probability of being selected for reproduction (Pfeifer and Scheier 2001, p.650).

Fitness Landscape A representation of how mutations can change the fitness of one or more organisms. If high fitness corresponds to high locations in the landscape, and if changes in genetic material are mapped to movements in the landscape, then evolution will tend to make populations move in an uphill direction on the fitness landscape (Flake 1998, p. 451).

Foraging Behaviour associated with the harvesting of food. It includes searching, recognising, handling, and consuming (Pfeifer and Scheier 2001, p.650).

Gene Pool All the genes in a population at a particular time (Ridley 2004, p.684).

Gene Sequence of nucleotides coding for a protein or, in some cases, part of a protein (Ridley 2004, p.684). A unit of heredity located on a chromosome and composed of DNA. The code necessary to promote the synthesis of one polypeptide chain (Fogel 2006, p.264).

Genetic Algorithm A method simulating the action of evolution within a computer. A population of fixed-length strings is evolved with a GA by employing crossover and mutation operators along with a fitness function that determines how likely individuals are to reproduce. GAs perform a type of search in a fitness landscape (Flake 1998, p. 452).

Genetic Drift Random changes in gene frequencies in a population (Ridley 2004, p.684).

Genetic Programming A method of applying simulated evolution on programs or program fragments. Modified forms of mutation and crossover are used along with a fitness function (Flake 1998, p.452).

Genome The entire collection of genetic materials; the totality of the genes possessed by an organism. The genome consists of one or more chromosomes that contain the individual genes (Pfeifer and Scheier 2001, p.650).

Genotype The set of two genes at a locus possessed by an individual (Ridley 2004, p.685). The set of genes of an individual (Mayr 2002, p.314). Refers to the particular set of genes contained in a genome, that is, an individual's genetic constitution (Pfeifer and Scheier 2001, p.651).

Grammar A formal language is defined as a set of strings where each string is a sequence of symbols. All the languages we are interested in consist of an infinite set of strings, so we need a concise way to characterise the set. We do that with a grammar (Russell and Norvig 2003, p.984).

Inheritable Refers to a trait that can be genetically passed from parent to offspring (Flake 1998, p.455).

Intron The nucleotide sequences of some genes consist of parts that code for amino acids, and other parts interspersed among them that do not code for amino acids. The interspersed non-coding parts, which are not translated, are called introns; the coding parts are called exons (Ridley 2004, p.685).

Kin Selection Selective advantage due to the altruistic interaction of individuals sharing part of the same genotype, such as siblings (Mayr 2002, p.315).

Lamarckism Evolutionary hypothesis that proposes the inheritance of acquired traits (Pfeifer and Scheier 2001, p.652).

Locus The location in the DNA occupied by a particular gene (Ridley 2004, p.686).

Mutation (Biology) When parental DNA is copied to form a new DNA molecule, it is normally copied exactly. A mutation is any change in the new DNA molecule from the parental DNA molecule. Mutations may alter single bases, or nucleotides, short stretches of bases, or parts of or whole chromosomes. Mutations can be detected both at the DNA level o the phenotypic level (Ridley 2004, p.686).

Mutation (Genetic Algorithm) Mutation takes the form of a "bit flip" with a fixed probability (De Jong 2006, p.26).

Natural Selection The process by which the forms of organisms in a population that are best adapted to the environment increase in frequency relative to less well adapted forms over a number of generations (Ridley 2004, p.686).

Neo-Darwinism Darwin's theory of natural selection plus Mendelian inheritance. The larger body of evolutionary thought that was inspired by the unification of natural selection and Mendelim. A synonym of modern synthesis (Ridley 2004, p.686).

NetLogo NetLogo is a programmable modelling environment for simulating natural and social phenomena. It was authored by Uri Wilensky in 1999 and has been in continuous development ever since at the Center for Connected Learning and Computer-Based Modeling (Wilensky, 1999).

Neutral Mutation Mutation with the same fitness as the other allele (or alleles) at its locus (Ridley 2004, p.686).

Phenotype The characters of an organism, whether due to the genotype or environment (Ridley 2004, p.687).

Population A group of organisms, usually a group of sexual organisms that interbreed and share a gene pool (Ridley 2004, p.687).

Protein A molecule made up of a sequence of amino acids. Many of the important molecules in a living thing are proteins; all enzymes, for example, are proteins (Ridley 2004, p.687).

Search Space A characterisation of every possible solution to a problem instance. For a genetic algorithm, it is every conceivable value assignment to the string in the population (Flake 1998, p.463).

Variation Genetic differences among individuals in a population (Flake 1998, p. 467).

References

Adamidis P. (1994) "Review of Parallel Genetic Algorithms Bibliography". Internal Technical Report, Faculty of Engineering, Aristotle University of Thessaloniki, November 1994.

Alba, E. ed. (2005) Parallel Metaheuristics: A New Class of Algorithms. USA: Wiley-Interscience.

Alba, E. and Tomassini, M. (2002) "Parallelism and Evolutionary Algorithms". IEEE Transactions on Evolutionary Computation 6(5), 443-462.

Alba, E. and Troya, J. M. (2001) "Gaining New Fields of Application for OOP: the Parallel Evolutionary Algorithm Case". Journal Of Object Oriented Programming, Dec 2001.

Amarteifio, S. (2005) Interpreting a Genotype-Phenotype Map with Rich Representations in XMLGE. MSc thesis, Department of Computer Science and Information Systems, University of Limerick, Ireland.

Amarteifio, S. and O'Neill, M. (2004) "An Evolutionary Approach to Complex System Regulation Using Grammatical Evolution". In Proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems (Artificial Life IX), pp.551-556, Boston, USA, 12-15 September 2004. MIT Press.

Amarteifio, S. and O'Neill, M. (2005) "Coevolving Antibodies with a Rich Representation of Grammatical Evolution". In Proceedings of the IEEE Congress on Evolutionary Computation 2005 (CEC '05), vol.1, pp.904-911, Edinburgh, UK, 2-5 September 2005. IEEE Press.

Angeline, P. J. and Pollack, J. B. (1993) "Evolutionary Module Acquisition". In Proceedings of the Second Annual Conference on Evolutionary Programming, pp. 154-163. San Diego, CA: Evolutionary Programming Society.

Angeline, P. J. and Pollack, J. B. (1994). "Coevolving High-Level Representations". In Proceedings of Artificial Life III, pp. 55-71. Reading, MA: Addison-Wesley.

Azad, A. (2003) A Position Independent Representation for Evolutionary Automatic Programming Algorithms – The Chorus System. PhD thesis, University of Limerick, Ireland.

Banzhaf, W. (1994) "Genotype-Phenotype-Mapping and Neutral Variation – A case study in Genetic Programming". In Proceedings of the International Conference on Evolutionary Computation, The Third Conference on Parallel Problem Solving from Nature (PPSN III), pp.322-332. London, UK: Springer-Verlag.

Barricelli, N. A. (1957) "Symbiogenetic Evolution Processes Realized by Artificial Methods". Methodos 9(35-36), 143-182.

Baum, E. B. (2004) What is Thought?. Cambridge, MA: MIT Press.

Blum, M. and Kozen, D. (1978) "On the power of the compass (or, why mazes are easier to search than graphs)". In Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1978), Ann Arbor, USA, 132-142.

Box, G. E. P. (1957) "Evolutionary Operation: A Method for Increasing Industrial Productivity". Applied Statistics, vol. 6, 81-101.

Brabazon, A. and O'Neill, M. (2006) Biologically Inspired Algorithms for Financial Modelling. Springer.

Byrne, J., Fenton, M., Hemberg, E., McDermott, J., O'Neill, M., Shotton, E. and Nally, C. (2011) "Combining Structural Analysis and Multi-Objective Criteria for Evolutionary Architectural Design". In Proceedings of the 9th European Event on Evolutionary and Biologically Inspired Music, Sound, Art and Design (EvoMUSART 2011), Lecture Notes in Computer Science, vol. 6625, pp.204-213, Torino, Italy, 27-29 April 2011. Berlin, Germany: Springer.

Byrne, J., O'Neill, M., McDermott, J. and Brabazon, A. (2009) "Structural and Nodal Mutation in Grammatical Evolution". In Proceedings of the Genetic and Evolutionary Computation Conference 2009 (GECCO '09), pp.1881-1882, Montreal, Québec, Canada, 8-12 July 2009. New York, NY, USA: ACM.

Cantú-Paz, E. (1998) "A Survey of Parallel Genetic Algorithms". Calculateurs Parallèles, Réseaux et Systèmes Répartis 10 (2), 141-171.

Cantú-Paz, E. (2001) Efficient and Accurate Parallel Genetic Algorithms. Massachusetts, USA: Kluwer Academic Publishers.

Cleary, R. (2005) Extending Grammatical Evolution with Attribute Grammars: An Application to Knapsack Problems. MSc thesis, University of Limerick, Ireland.

Cleary, R. and O'Neill, M. (2005) "An Attribute Grammar Decoder for the 01 Multiconstrained Knapsack Problem". In Proceedings of the European Conference on Evolutionary Combinatorial Optimisation 2005 (EvoCOP '05), Lecture Notes in Computer Science, vol. 3448, pp.34-45, Lausanne, Switzerland, 30 March - 1 April 2005. Berlin, Germany: Springer.

Cramer, N. L., (1985). "A Representation for the Adaptive Generation of Simple Sequential Programs". In Proceedings of the First International Conference on Genetic Algorithms and Their Applications, pp.183-187, Carnegie-Mellon University, Pittsburgh, USA, 24-26 July 1985. Hillsdale, NJ: Lawrence Erlbaum Associates.

Cui, W., Brabazon, A. and O'Neill, M. (2010) "Evolving Dynamic Trade Execution Strategies using Grammatical Evolution". In Proceedings of the 4th European Event on Evolutionary and Natural Computation in Finance and Economics (EvoFin 2010), Applications of Evolutionary Computation, Lecture Notes in Computer Science, vol. 6025, pp.192-201, Istanbul, Turkey, 7-9 April 2010. Berlin, Germany: Springer.

Darwin, C. (1859) On the Origin of Species by Means of Natural Selection or the Preservations of Favored Races in the Struggle for Life. London, Great Britain: John Murray.

De Jong, K. A. (2006) Evolutionary Computation: A Unified Approach. Cambridge, MA: MIT Press.

Dempsey, I., O'Neill, M. and Brabazon, A. (2005) "meta-Grammar Constant Creation with Grammatical Evolution by Grammatical Evolution". In Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO '05), pp. 1665-1671. New York, USA: ACM Press.

Dempsey, I., O'Neill, M. and Brabazon, A. (2006) "Adaptive Trading with Grammatical Evolution". In Proceedings of the 2006 IEEE Congress on Evolutionary Computation, pp. 2587-2592.

Dempsey, I., O'Neill, M. and Brabazon, A. (2009) Foundations in Grammatical Evolution for Dynamic Environments. Berlin, Germany: Springer.

Flake, G. W. (1998) The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation. Cambridge, USA: MIT Press.

Fogel, D. B. (2006) Evolutionary Computation: Toward a New Philosophy of Machine Intelligence, 3rd ed. New Jersey, USA: IEEE Press.

Fogel, L., Owens, A. and Walsh, M. (1966) Artificial Intelligence through Simulated Evolution. New York: John Wiley & Sons.

Friedberg, R. M. (1958) "A Learning Machine: Part I". IBM Journal of Research and Development, vol. 2, 2-13.

Friedman, G. (1956) Select feedback computers for engineering synthesis and nervous system analogy. Master's thesis, UCLA.

Gagné, C., Schoenauer, M., Parizeau, M. and Tomassini, M. (2006) "Genetic Programming, Validation Sets, and Parsimony Pressure". In Proceedings of the 9th European Conference on Genetic Programming (EuroGP '06), Lecture Notes in Computer Science, vol. 3905, pp.109–120, Budapest, Hungary, 10-12 April 2006. Berlin, Germany: Springer.

Galván-López , E., Swafford, J. M., O'Neill, M. and Brabazon, A. (2010) "Evolving a Ms. PacMan Controller using Grammatical Evolution". In Proceedings of Applications of Evolutionary Computation, EvoApplicatons 2010: EvoGAMES, Lecture Notes in Computer Science, vol. 6024, pp.161-170, Istanbul, Turkey, 7-9 April 2010. Berlin, Germany: Springer.

Gärdenfors, P. (2004) Conceptual Spaces: The Geometry of Thought. Cambridge, MA: MIT Press.

Garibay, Ö. Ö. (2008) Analyzing the Effects of Modularity on Search Spaces. PhD thesis, School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida.

Gavrilis, D., Tsoulos, I. G., Georgoulas, G. and Glavas, E. (2005) "Classification of fetal heart rate using grammatical evolution". In Proceedings of the 2005 IEEE Workshop on Signal Processing Systems Design and Implementation, 425-429.

Georgiou, L. (2006) Java GE (jGE) Official Web Site. Artificial Intelligence and Intelligent Agents Research Group, School of Computer Science, Bangor University, Available from http://aiia.bangor.ac.uk/jge.

Georgiou, L. and Teahan, W. J. (2006a) "jGE – A Java implementation of Grammatical Evolution". 10th WSEAS International Conference on Systems, Athens, Greece, July 10-15, 2006.

Georgiou, L. and Teahan, W. J. (2006b) "Implication of Prior Knowledge and Population Thinking in Grammatical Evolution: Toward a Knowledge Sharing Architecture". WSEAS Transactions on Systems 5 (10), 2338-2345.

Georgiou, L. and Teahan, W. J. (2008) "Experiments with Grammatical Evolution in Java". Knowledge-Driven Computing: Knowledge Engineering and Intelligent Computations, Studies in Computational Intelligence (vol. 102), 45-62. Berlin, Germany: Springer Berlin / Heidelberg.

Georgiou, L. and Teahan, W. J. (2010) "Grammatical Evolution and the Santa Fe Trail Problem". In Proceedings of the International Conference on Evolutionary Computation (ICEC 2010), October 24-26, 2010, Valencia, Spain, 10-19.

Georgiou, L. and Teahan, W. J. (2011) "Constituent Grammatical Evolution". In Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI 2011), July 16-22, 2011, Barcelona, Catalonia (Spain), 1261-1268.

Ghanea-Hercock, R. (2003) Applied Evolutionary Algorithms in Java. New York, NY: Springer.

Harper, R. (2010) "GE, Explosive Grammars and the Lasting Legacy of Bad Initialisation". In Proceedings of the IEEE Congress on Evolutionary Computation 2010 (CEC '10), pp.1-8, Barcelona, Spain, 18-23 July 2010. IEEE Press.

Harper, R. (2011) "Co-evolving robocode tanks". In Proceedings of the Genetic and Evolutionary Computation Conference 2011 (GECCO '11), pp. 1443-1450, Dublin, Ireland, 12-16 July 2011. New York, NY, USA: ACM.

Harper, R. and Blair, A. (2005) "A Structure Preserving Crossover in Grammatical Evolution". In Proceedings of the 2005 IEEE Congress on Evolutionary Computation, vol. 3, 2537-2544.

Harper, R. and Blair, A. (2006a) "A Self-Selecting Crossover Operator". In Proceedings of the 2006 IEEE Congress on Evolutionary Computation (IEEE CEC 2006), pp. 1420-1427.

Harper, R. and Blair, A. (2006b) "Dynamically Defined Functions In Grammatical Evolution". In Proceedings of the 2006 IEEE Congress on Evolutionary Computation (IEEE CEC 2006), pp. 2638-2645.

Hemberg, E. (2010) An Exploration of Grammars in Grammatical Evolution. PhD thesis, School of Computer Science Department and Informatics, University College Dublin, Ireland.

Hemberg, E., Ho, L., O'Neill, M. and Claussen, H. (2011) "A Symbolic Regression Approach To Manage Femtocell Coverage using Grammatical Genetic Programming". In Companion Material Proceedings of the Genetic and Evolutionary Computation Conference 2011 (GECCO '11), pp.639-646, Dublin, Ireland, 12-16 July 2011. New York, NY, USA: ACM.

Hemberg, E., O'Neill, M. and Brabazon, A. (2008) "Grammatical Bias and Building Blocks in Meta-Grammar Grammatical Evolution". In Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2008, 3775-3782.

Hemberg, E., O'Neill, M. and Brabazon, A. (2009) "An investigation into automatically defined function representations in Grammatical Evolution". In Proceedings of the 15th International Conference on Soft Computing, Mendel '09, Brno, Czech Republic, 24-26 June 2009.

Hirsh, H. (2000) "Trends & Controversies: Genetic programming". IEEE Intelligent Systems 15(3), 74-84.

Holland, J. (1962) "Outline for a Logical Theory of Adaptive Systems". JACM 9, 297-134.

Holland, J. (1967) "Nonlinear environments permitting efficient adaptation". In Computer and Information Sciences II. Academic Press.

Holland, J. (1975) Adaptation in Natural and Artificial Systems. Ann Arbor, MI: University of Michigan Press.

Hugosson, J., Hemberg, E., Brabazon, A. and O'Neill, M. (2010) "Genotype Representations in Grammatical Evolution". Applied Soft Computing 10(1), 36-43.

IBM Corporation (2004), Jikes 1.22 (online). United States: NY. Available from: http://jikes.sourceforge.net (Accessed 27 March 2006).

Jefferson, D., Collins, R., Cooper, C., Dyer, M, Flowers, M., Korf, R., Taylor, C. and Wang, A. (1992) "Evolution as a Theme in Artificial Life: The Genesys/Tracker System". In Langton, Christopher, et al. (editors), Artificial Life II. Addison-Wesley.

Joshi, A. (1985) "Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions?". Natural Language Parsing, pp.205–250. New York: Cambridge University Press.

Joshi, A. and Schabes, Y. (1997) Tree-Adjoining Grammars. Handbook of Formal Languages, Beyond Words, vol.3, pp.69–123.

JUnit.org (2006) JUnit – Testing Resources for Extreme Programming (online). Available from: http://www.junit.org (Accessed 11 February 2006).

Keijzer, M., Babovic, V., Ryan, C., O'Neill, M. and Cattolico, M. (2001) "Adaptive Logic Programming". In Proceedings of the Genetic and Evolutionary Computation Conference 2001 (GECCO '01), pp.42-49, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

Keijzer, M., Ryan, C. and Cattolico, M. (2004) "Run Transferable Libraries - Learning Functional Bias in Problem Domains". In Proceedings of the Genetic and Evolutionary Computation Conference 2004 (GECCO '04), Part II, Lecture Notes in Computer Science (vol. 3103), pp.531-542, Seattle, Washington, USA, 26-30 June 2004. Springer.

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

Koza, J. R. (1994) Genetic Programming II: Automatic Discovery of Reusable Programs. Cambridge, MA: MIT Press.

Koza, J. R., Bennet III, F. H., Andre, D. and Keane M. A. (1999) Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann Publishers.

Koza, J. R., Keane, M. A., Streeter, M. J., Mydlowec, W., Yu, J. and Lanza, G. (2003) Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers.

Langdon, W. B. and Poli, R. (1998a) "Why Ants are Hard". In Koza et al. (editors), Genetic Programming 1998: Proceedings of the Third Annual Conference, pp.193-201. San Francisco, California: Morgan Kaufmann.

Langdon, W. B. and Poli, R. (1998b) Better Trained Ants for Genetic Programming. Technical report, University of Birmingham, School of Computer Science (CSRP-98-12), April 1998.

Langdon, W. B. and Poli, R. (2001) Foundations of Genetic Programming. Berlin: Springer.

Luke, S. and Wiegand, R. P. (2002) "When Coevolutionary Algorithms Exhibit Evolutionary Dynamics". In Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference 2002 (GECCO '02), pp.236-241, New York, USA, 8 July 2002. New York, NY, USA:AAAI Press.

Majeed, H. and Ryan, C. (2007) "Context-Aware Mutation: A Modular, Context Aware Mutation Operator for Genetic Programming". In Proceedings of the Genetic and

Evolutionary Computation Conference 2007 (GECCO '07), pp.1651-1658, London, England, 7-11 July 2007. New York, NY, USA: ACM.

Mayr, E. (2002) What Evolution Is. London, Great Britain: Phoenix.

McDermott, J., Byrne, J., Swafford, J. M., O'Neill, M. and Brabazon, A. (2010) "Higher-Order Functions in Aesthetic EC Encodings". In Proceedings of the IEEE Congress on Evolutionary Computation 2010 (CEC '10), pp.1-8, Barcelona, Spain, 18-23 July 2010. IEEE Press.

McGee, R., O'Neill, M. and Brabazon, A. (2010) "The Syntax of Stock Selection: Grammatical Evolution of a Stock Picking Model". In Proceedings of the IEEE Congress on Evolutionary Computation 2010 (CEC '10), pp.1-8, Barcelona, Spain, 18-23 July 2010. IEEE Press.

McKay, R. I., Hoai, N. X., Whigham, P. A., Shan, Y. and O'Neill M. (2010) "Grammar-based Genetic Programming: a survey". Genetic Programming and Evolvable Machines 11(3-4), 365-396.

McKay, R. I., Hoang, T. H., Essam, D., and Nguyen, X. H. (2006) "Developmental Evaluation in Genetic Programming: the Preliminary Results". In Proceedings of the 9th European Conference on Genetic Programming (EuroGP '06), Lecture Notes in Computer Science, vol. 3905, pp.280–289, Budapest, Hungary, 10-12 April 2006. Berlin, Germany: Springer.

Murphy, E. (2011) "Examining Grammars and Grammatical Evolution in Dynamic Environments". In Proceedings of the Genetic and Evolutionary Computation Conference 2011 (GECCO '11), pp.779-782, Dublin, Ireland, 12-16 July 2011. New York, NY, USA: ACM.

Murphy, J. E. (2011) Applications of Evolutionary Computation to Quadrupedal Animal Animation. PhD thesis, School of Computer Science and Informatics, University College Dublin, Ireland.

Murphy, J. E., Carr, H. and O'Neill M. (2010) "Animating Horse Gaits and Transitions". EG UK Theory and Practice of Computer Graphics 2010, University of Sheffield, UK, 6-8 September 2010. Eurographic Digital Library.

Murphy, J. E., O'Neill, M. and Carr, H. (2009) "Exploring Grammatical Evolution for Horse Gait Optimisation". In Proceedings of the 12th European Conference on Genetic Programming (EuroGP 2009), Lecture Notes in Computer Science, vol. 5481, pp.183–194, Tübingen, Germany, 15-17 April, 2009. Springer.

Murphy, E., O'Neill, M., Galván-López, E. and Brabazon, A. (2010) "Tree-Adjunct Grammatical Evolution". In IEEE Congress on Evolutionary Computation 2010 (IEEE CEC '10), pp.4449–4456, Barcelona, Spain, 18-23 July 2010. IEEE Press.

Nelson, M. (2001) Robocode. Available from http://robocode. sourceforge.net.

Nicolau, M. (2005), libGE: Grammatical Evolution Library for version 0.24, 20 October 2005. Available from http://waldo.csisdmz.ul.ie/ libGE/libGE.pdf (Accessed 30 November 2005).

Nicolau, M. (2006a), libGE: Grammatical Evolution Library for version 0.26beta1, 3 March 2006. Available from http://waldo.csisdmz.ul.ie/ libGE/libGE.pdf (Accessed 11 March 2006).

Nicolau, M. (2006b), libGE: Grammatical Evolution Library for version 0.27alpha1, 14 September 2006. Available from: http://bds.ul.ie/ libGE/libGE.pdf (Accessed 20 October 2008).

Nicolau, M. and Costelloe, D. (2011) "Using Grammatical Evolution to Parameterise Interactive 3D Image Generation". In Proceedings of the 2011 International Conference on Applications of Evolutionary Computation (EvoApplications 2011), Lecture Notes in Computer Science, vol. 6625, pp. 374–383, Torino, Italy, 27-29 April 2011. Berlin, Germany: Springer.

Nicolau, M. and Dempsey, I. (2006) "Introducing Grammar Based Extensions for Grammatical Evolution". In Proceedings of the 2006 IEEE Congress on Evolutionary Computation (IEEE CEC 2006), pp. 648-655.

Nicolau, M. and Ryan, C. (2002) LINKGAUGE: Tackling hard deceptive problems with a new linkage learning genetic algorithm. Available from http://www.grammatical-evolution.org/papers/ nicolau02linkgauge.ps (Accessed 14 December 2005).

Nicolau, M. and Ryan, C. (2003) "How functional dependency adapts to salience hierarchy in the GAuGE system". In Proceedings of EuroGP 2003. Available from http://www.grammatical-evolution.org/papers/ nicolau03salience.ps (Accessed 14 December 2005).

O'Neill, M. (1999) "Automatic Programming with Grammatical Evolution". In Proceedings of the Genetic and Evolutionary Computation Conference Workshop Program held in Orlando, Florida USA 13-17 July 1999, San Francisco, CA: Morgan Kaufmann. Available from http://www.grammatical-evolution.org/papers/gradworkshop99.ps (Accessed 30 November 2005).

O'Neill, M. and Brabazon, A. (2005a) "Recent Adventures in Grammatical Evolution". In Proceedings of Computer Methods and Systems Conference (CMS '05), vol. 1, pp.245-253, Krakow, Poland, 14-16 November 2006. Poland: Oprogramowanie Naukowo-Techniczne Tadeusiewicz.

O'Neill, M. and Brabazon, A. (2005b) "mGGA: The meta-Grammar Genetic Algorithm". Genetic Programming, Lecture Notes in Computer Science, vol. 3447, pp.311-320. Berlin, Germany: Springer Berlin / Heidelberg.

O'Neill M. and Brabazon, A. (2006a) "Grammatical Swarm: The generation of programs by social programming". Natural Computing 5(4), 443-462.

O'Neill, M. and Brabazon, A. (2006b) "Grammatical Differential Evolution". In Proceedings of the 2006 International Conference on Artificial Intelligence (ICAI '06), vol. 1, pp. 231-236, Las Vegas, Nevada, USA, 26-29 June 2006. CSEA Press.

O'Neill, M. and Brabazon, A. (2008) "Evolving a Logo Design Using Lindenmayer Systems, Postscript and Grammatical Evolution". 2008 IEEE World Congress on Computational Intelligence, 3788-3794. Hong Kong: IEEE Press.

O'Neill, M. and Brabazon, A. (2009) "Recent Patents on Genetic Programming". Recent Patents on Computer Science 2(1), 43-49.

O'Neill, M., Brabazon, A. and Adley, C. (2004) "The Automatic Generation of Programs for Classification Problems with Grammatical Swarm". In Proceedings of the 2004 Congress on Evolutionary Computation, vol. 1, 104 – 110.

O'Neill, M., Brabazon, A., Nicolau, M., Garraghy, S. M. and Keenan P. (2004) "πGrammatical Evolution". In Proceedings of the Genetic and Evolutionary Computation Conference 2004 (GECCO '04), Part II, Lecture Notes in Computer Science, vol. 3103, pp.617-629, Seattle, Washington, USA, 26-30 June 2004. Springer.

O'Neill, M., Brabazon, A., Ryan, C. and Collins, J. J. (2001a) "Evolving Market Index Trading Rules using Grammatical Evolution". In Proceedings of EvoIASP 2001. Available from http:// www.grammatical-evolution.org/papers/evoiasp2001.ps.gz (Accessed 14 December 2005).

O'Neill, M., Brabazon, A., Ryan, C. and Collins, J. J. (2001b) "Developing a Market Timing System using Grammatical Evolution". In Proceedings of GECCO 2001. Available from http://www.grammatical-evolution.org/papers/gecco_iseq2001.ps.gz (Accessed 14 December 2005).

O'Neill, M., Cleary, R. and Nikolov, N. (2004) "Solving Knapsack Problems with Attribute Grammars". In Proceedings of the Third Grammatical Evolution Workshop 2004 (GEWS '04), Seattle, Washington, USA, 26-30 June 2004.

O'Neill, M. and Ryan, C. (1999a) "Automatic Generation of Caching Algorithms". In Proceedings of EUROGEN 1999, Short Course on Evolutionary Algorithms in Engineering and Computer Science held in Jyväskylä, Finland 30 May - 3 June 1999, pages 127-134. Available from http://www.grammatical-evolution.org/papers/eurogen99.ps.gz (Accessed 30 November 2005).

O'Neill, M. and Ryan, C. (1999b) "Automatic Generation of High Level Functions using Evolutionary Algorithms". In Proceedings of SCASE 1999, Soft Computing and Software Engineering Workshop held in University of Limerick, Ireland 1999. Available from http://www.grammatical-evolution.org/papers/scase99.ps.gz (Accessed 30 November 2005).

O'Neill, M. and Ryan, C. (1999c) "Automatic Generation of Programs with Grammatical Evolution". In Proceedings of AICS 1999, pp. 72-78. Available from http://www.grammatical-evolution.org/papers/ aics99.ps.gz (Accessed 30 November 2005).

O'Neill, M. and Ryan, C. (1999d) "Evolving Multi-line Compilable C Programs". In Proceedings of the Second European Workshop on Genetic Programming, 1999, pp. 83-92. Available from http:// www.grammatical-evolution.org/papers/eurogp99.ps.gz (Accessed 30 November 2005).

O'Neill, M. and Ryan, C. (1999e) "Genetic Code Degeneracy: Implications for Grammatical Evolution and Beyond". In Proceedings of the European Conference on Artificial Life 1999. Available from http://www.grammatical-evolution.org/papers/ecal99.ps.gz (Accessed 30 November 2005).

O'Neill, M. and Ryan, C. (1999f) "Under the Hood of Grammatical Evolution". In GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference held in Orlando, Florida, USA 13-17 July 1999, edited by W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, 1999. San Francisco, CA: Morgan Kaufmann. Available from http://www.grammatical-evolution.org/papers/GP-434.ps.gz (Accessed 30 November 2005).

O'Neill, M. and Ryan, C. (2000a) "Crossover in Grammatical Evolution: A Smooth Operator?". In Proceedings of the Third European Workshop on Genetic Programming 2000, pp. 149-162. Available from http://www.grammatical-evolution.org/papers/eurogp2000.ps.gz (Accessed 14 December 2005).

O'Neill, M. and Ryan, C. (2000b) "Grammar based function definition in Grammatical Evolution". In Proceedings of GECCO 2000, the Genetic and Evolutionary Computation Conference, pp. 485-490. Available from http://www.grammatical-evolution.org/papers/gecco2000.ps.gz (Accessed 14 December 2005).

O'Neill, M. and Ryan, C. (2000c) "Incorporating Gene Expression Models into Evolutionary Algorithms". In Proceedings of the 2000 Genetic and Evolutionary Computation Conference Workshop Program, pp. 167-172. Available from http://www.grammatical-evolution.org/papers/gecco2000_gemworkshop.ps.gz (Accessed 14 December 2005).

O'Neill, M. and Ryan, C. (2001) "Grammatical Evolution". IEEE Transactions on Evolutionary Computation 5(4), 349-358.

O'Neill, M. and Ryan, C. (2003) Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. USA: Kluwer Academic Publishers.

O'Neill, M. and Ryan, C. (2004) "Grammatical Evolution by Grammatical Evolution: The Evolution of Grammar and Genetic Code". In Proceedings of the 7th European Conference (EuroGP 2004), pp. 138-149. Berlin, Germany: Springer Berlin / Heidelberg.

O'Neill, M., Ryan, C., Keijzer, M. and Cattolico, M. (2001) "Crossover in Grammatical Evolution: The Search Continues". In Proceedings of EuroGP 2001. Available from http:// www.grammatical-evolution.org/papers/eurogp2001.ps.gz (Accessed 14 December 2005).

O'Neill, M., Ryan, C., Keijzer, M. and Cattolico, M. (2003) "Crossover in Grammatical Evolution". Genetic Programming and Evolvable Machines 4(1), 67-93.

O'Neill, M., Ryan, C. and Nicolau, M. (2001) "Grammar Defined Introns: An Investigation into Grammars, Introns, and Bias in Grammatical Evolution". In Proceedings of GECCO 2001. Available from http://www.grammatical-evolution.org/papers/gecco2001.ps.gz (Accessed 14 December 2005).

O'Neill, M., Swafford, J., McDermott, J., Byrne, J., Brabazon, A., Shotton, E., McNally, C. and Hemberg, M. (2009) "Shape grammars and grammatical evolution for evolutionary design". In Proceedings of Genetic and Evolutionary Computation Conference.

O'Neill, M., Vanneschi, L., Gustafson, S. and Banzhaf, W. (2010) "Open issues in genetic programming". Genetic Programming and Evolvable Machines 11(3-4), 339-363.

O'Sullivan, J. and Ryan, C. (2002) "An investigation into the use of different search strategies with grammatical evolution". In Proceedings of the 5th European Conference on Genetic Programming (EuroGP '02), Lecture Notes in Computer Science, vol. 2278, pp.268-277, Kinsale, Ireland, 3-5 April 2002. Berlin, Germany: Springer.

Ortega, A., Cruz, M. and Alfonseca, M. (2007) "Christiansen Grammar Evolution: Grammatical Evolution with Semantics". IEEE Transactions on Evolutionary Computation 11(1), 77-90.

Ošmera, P., Popelka, O. and Pivoňka, P. (2006) "Parallel Grammatical Evolution with Backward Processing". In Proceedings of the 9th International Conference on Control, Automation, Robotics and Vision (ICARCV 2006), pp. 1-6.

Paterson, N. and Livesey, M. (1997) "Evolving caching algorithms in C by GP". In Genetic Programming 1997, pages 262-267. MIT Press.

Perez, D., Nicolau, M., O'Neill, M. and Brabazon, A. (2011) "Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution". In Proceedings of EvoGAMES 2011 the 3rd European Event on Bio-inspired Algorithms in Games, Lecture Notes in Computer Science, vol. 6624, pp.123-132, Torino, Italy, 27-29 April 2011. Berlin, Germany: Springer.

Pfeifer, R. and Scheier, C. (2001) Understanding Intelligence. Cambridge, MA: MIT Press.

Rechenberg, I. (1965) "Cybernatic Solution Path of an Experimental Problem". In Library Translation 1122. Farnborough: Royal Aircraft Establishment.

Ridley, M. (2004) Evolution, 3rd ed. Great Britain: Blackwell Publishing.

Robilliard, D., Mahler, S., Verhaghe, D. and Fonlupt, C. (2006) "Santa Fe Trail Hazards". In Proceedings of the 7th International Conference on Artificial Evolution (EA '05), Lecture Notes in Computer Science, vol. 3871, pp.1-12, Lille, France, 26-28 October 2005. Berlin, Germany: Springer, 2006.

Rosca, J. P. and Ballard, D. H. (1994) "Genetic Programming with Adaptive Representations". Technical Report 489, University of Rochester, Rochester, NY, USA, 1994.

Rothlauf, F. and Oetzel, M. (2006) "On the Locality of Grammatical Evolution". In Proceedings of the 9th European Conference on Genetic Programming, Lecture Notes in Computer Science, vol. 3905, pp.320-330, Budapest, Hungary. Berlin, Germany: Springer.

Russell S. and Norvig P. (2003) Artificial Intelligence: A Modern Approach, 2nd ed. USA: Prentice Hall.

Ryan, C., Azad, A., Sheahan, A. and O'Neill, M. (2002) "No Coercion and No Prohibition, A Position Independent Encoding Scheme for Evolutionary Algorithms – The Chorus System". In Proceedings of the 5th European Conference on Genetic Programming (EuroGP '02), Lecture Notes in Computer Science, vol. 2278, pp.131–141, Kinsale, Ireland, 3-5 April 2002. Berlin, Germany: Springer.

Ryan, C., Collins, J. J. and O'Neill, M. (1998) "Grammatical Evolution: Evolving Programs for an Arbitrary Language". Lecture Notes in Computer Science, vol. 1391. First European Workshop on Genetic Programming 1998. Available from http://www.grammatical-evolution.org/papers/eurogp98.ps (Accessed 30 November 2005).

Ryan, C., Keijzer, M. and Cattolico, M. (2004) "Favourable Biasing of Function Sets Using Run Transferable Libraries". In O'Reilly, M., Yu, T. and Riolo, R. (Eds) Genetic Programming, Theory and Practice II, R. Riolo et al (eds.), vol. 8, pp.103-120. Michigan, USA: University of Michigan Press.

Ryan, C. and O'Neill, M. (1998) "Grammatical Evolution: A Steady State Approach". In Proceedings of the Second International Workshop on Frontiers in Evolutionary Algorithms, 1998, pp. 419-423. Available from http://www.grammatical-evolution.org/papers/fea98.ps (Accessed 30 November 2005).

Ryan, C., O'Neill, M. and Azad A. (2001) "No Coercion and No Prohibition – A Position Independent Encoding Scheme for Evolutionary Algorithms". In Proceedings of the Genetic and Evolutionary Computation Conference 2001 (GECCO '01), p.187, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

Ryan, C., O'Neill, M. and Collins, J. J. (1998) "Grammatical Evolution: Solving Trigonometric Identities". In Proceedings of Mendel 1998: 4th International Mendel Conference on Genetic Algorithms, Optimisation Problems, Fuzzy Logic, Neural Networks, Rough Sets held in Brno, Czech Republic June 24-26 1998, pp. 111-119. Available from http://www.grammatical-evolution.org/papers/mendel98.ps (Accessed 30 November 2005).

Shao, J., McDermott, J., O'Neill, M. and Brabazon, A. (2010) "Jive: A Generative, Interactive, Virtual, Evolutionary Music System". Applications of Evolutionary Computation, Lecture Notes in Computer Science, vol. 6025, pp.341-350. Berlin, Germany: Springer Berlin / Heidelberg.

Sondahl, F. (2005) Genetic Programming Library for NetLogo project, Northwestern University. Available from http://cs.northwestern.edu/ ~fjs750/netlogo/final.

Swafford, J. M., Hemberg, E., O'Neill, M., Nicolau, M. and Brabazon, A. (2011) "A Non-Destructive Grammar Modification Approach to Modularity in Grammatical Evolution". In Proceedings of the Genetic and Evolutionary Computation Conference 2011 (GECCO '11), pp.1411-1418, Dublin, Ireland, 12-16 July 2011. New York, NY, USA: ACM.

Swafford, J. M. and O'Neill, M. (2010) "An examination on the modularity of grammars in grammatical evolutionary design". In Proceedings of the IEEE Congress on Evolutionary Computation 2010 (CEC '10), pp.1-8, Barcelona, Spain, 18-23 July 2010. IEEE Press.

Swafford, J. M., O'Neill, M. and Nicolau, M. (2011) "Exploring Grammatical Modification with Modules in Grammatical Evolution". In Proceedings of the 14th European Conference (EuroGP '11), pp.310-321, Torino, Italy, 27-29 April 2011. Berlin, Germany: Springer.

Teahan, W. J. (2010a) Artificial Intelligence – Agents and Environments. Ventus Publishing ApS.

Teahan, W. J. (2010b) Artificial Intelligence – Agent Behaviour I. Ventus Publishing ApS.

Teahan, W. J., Al-Dmour, N. and Tuff, P. G. (2005) "On thought, knowledge, evolution and search". In Proceedings of Computer Methods and Systems CMS'05 Conference held in Krakow, Poland 14-16 November 2005.

Tsoulos, J. G., Gavrilis, D. and Glavas, E. (2005) "Neural network construction using grammatical evolution". In Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology 2005, 827-831.

UCD Natural Computing Research & Application Group (2008) Grammatical Evolution in Java (GEVA) Official Web Site. Ireland, Dublin. Available from http://ncra.ucd.ie/geva.

UCD Natural Computing Research & Application Group (2010) Grammatical Evolution in MATLAB (GEM) v0.2. Ireland, Dublin. Available from http://ncra.ucd.ie/GEM/GEM-v0.2.tgz.

Whigham, P. A. (1995a) "Grammatically-based Genetic Programming". In Proceedings of the 1995 Workshop on Genetic Programming: From Theory to Real-World Applications, pp. 33-41, Tahoe City, California, USA, 9 July 1995.

Whigham, P. A. (1995b) "Inductive Bias and Genetic Programming". In Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, pp. 461–466, Sheffield, UK, 12-14 September 1995.

Whigham, P. A. (1996) Grammatical Bias for Evolutionary Learning. PhD thesis, School of Computer Science, University College, University of New South Wales, Australian Defence Force Academy, Canberra, Australia.

White, B. C., Reif, D. M., Gilbert, J. C. and Moore, J. H. (2005) "A Statistical Comparison of Grammatical Evolution Strategies in the Domain of Human Genetics". In Proceedings of the 2005 IEEE Congress on Evolutionary Computation, vol. 1, 491 – 497.

Wilensky, U. (1999) NetLogo. Evanston, IL: Center for Connected Learning and Computer-Based Modeling, Northwestern University. Available from http://ccl.northwestern.edu/netlogo.

Wolpert, D. H. and Macready, W. G (1997) "No Free Lunch Theorems for Optimization". IEEE Transactions on Evolutionary Computation 1(1), 67-82.

Wright, S. (1932) "The roles of mutation, inbreeding, crossbreeding, and selection in evolution". In Proceedings of the 6th International Congress on Genetics, vol. 1, 356-366.