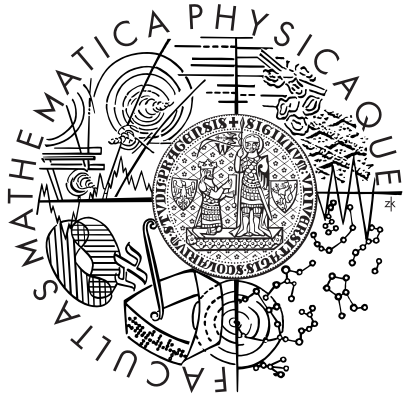
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Adam Nohejl

Grammar-based genetic programming

Department of Software and Computer Science Education

Supervisor of the master thesis: RNDr. František Mráz, CSc.

Study programme: Computer science
Specialisation: Theoretical computer science

Prague 2011

I would like to thank to František Mráz, who supervised my work on this thesis, for his guidance and helpful suggestions. I am also grateful to Dr Man Leung Wong for providing his implementation of the logic grammar based framework LOGENPRO.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

Prague, 25 July 2011 Adam Nohejl

Title: Grammar-based genetic programming

Author: Adam Nohejl

Department: Department of Software and Computer Science Education

Supervisor of the master thesis: RNDr. František Mráz, CSc.

Abstract: Tree-based genetic programming (GP) has several known shortcomings: difficult adaptability to specific programming languages and environments, the problem of closure and multiple types, and the problem of declarative representation of knowledge. Most of the methods that try to solve these problems are based on formal grammars. The precise effect of their distinctive features is often difficult to analyse and a good comparison of performance in specific problems is missing. This thesis reviews three grammar-based methods: context-free grammar genetic programming (CFG-GP), including its variant GPHH recently applied to exam timetabling, grammatical evolution (GE), and LOGENPRO, it discusses how they solve the problems encountered by GP, and compares them in a series of experiments in six applications using success rates and derivation tree characteristics. The thesis demonstrates that neither GE nor LOGENPRO provide a substantial advantage over CFG-GP in any of the experiments, and analyses the differences between the effects of operators used in CFG-GP and GE. It also presents results from a highly efficient implementation of CFG-GP and GE.

Keywords: genetic programming, formal grammar, evolutionary algorithms, grammatical evolution.

Název práce: Genetické programování založené na gramatikách

Autor: Adam Nohejl

Katedra: Kabinet software a výuky informatiky

Vedoucí diplomové práce: RNDr. František Mráz, CSc.

Abstrakt: Genetické programování (GP) založené na stromech má několik známých nedostatků: složité přizpůsobení specifickým programovacím jazykům a prostředím, problém uzávěru a více typů a problém deklarativní reprezentace vědomostí. Většina metod, které se snaží tyto problémy vyřešit, je založena na formálních gramatikách. Přesné důsledky vlastností, které je odlišují, je těžké analyzovat a dobré srovnání výsledků v konkrétních problémech chybí. Tato práce zkoumá tři metody založené na gramatikách: genetické programování s bezkontextovými gramatikami (CFG-GP), včetně jeho varianty GPHH nedávno aplikované na rozvrhování zkoušek, gramatickou evoluci (GE) a LOGENPRO, pojednává o tom, jak řeší problémy GP, a porovnává je v sérii experimentů v šesti aplikacích podle četností úspěchu a charakteristik derivačních stromů. Práce ukazuje, že GE ani LOGENPRO neposkytují podstatnou výhodu v žádném z experimentů a analyzuje rozdíly v účincích operátorů používaných v CFG-GP a GE. Jsou také prezentovány výsledky velmi efektivní implementace metod CFG-GP a GE.

Klíčová slova: genetické programování, formální gramatika, evoluční algoritmy, gramatická evoluce.

Contents

Introduction

This thesis explores and analyses methods of genetic programming based on formal grammars. *Genetic programming* (GP) is a metaheuristic for deriving problem solutions in the form of programs based on evolutionary principles, and thus belonging to the family of *evolutionary algorithms*. Traditionally, genetic programming was tied to the Lisp programming language, taking advantage of its simplicity and straightforward correspondence between a program and its parse tree. Several other strains of genetic programming were devised over time: *linear GP* systems departed from the original premise of the *tree-based GP* that programs are trees, and variants of tree-based GP usually either extended the original genetic programming with new features (such as automatically defined functions, which effectively add co-evolution of subroutines) or placed restrictions on permitted tree forms (such as strongly-typed genetic programming).

Various *grammar-based GP* methods, which have also emerged, either can be put in the same category with tree-based GP or linear GP, or often more appropriately, can be thought of as being in between them. For a computer scientist, a formal grammar is the natural link between program as a tree and program as a linear string. This is always the primary role of grammars in genetic programming: they specify the language of candidate solutions. The grammar-based GP methods, however, can be employed in several ways:

(1) Constraining tree-based GP: The grammar is used to restrict the search space, and consequently also to redefine search operators under which the restricted search space is algebraically closed.

(2) Introduction of bias into tree-based GP: The grammar is used as a vehicle for bias toward certain solutions. The bias may also be adjusted over the course of the algorithm's execution.

(3) Replacement of the traditional tree-based GP mechanisms: The grammar may serve both of the above purposes, but more importantly it is an integral part of the algorithm that provides mapping between two representations of candidate solutions.

From the short descriptions we can already glimpse that different grammar-based GP methods have different aims. In case (1) the grammar is used to remedy a shortcoming of GP: the so-called *closure* problem, but most grammar-based methods also raise problems and questions of their own concerning the encoding of individuals, and the design of operators. In spite of the different motivations behind the methods, there are also significant areas of overlap

between them. The goal of this thesis will therefore be to

- describe the problems arising from integration of grammars and genetic programming,
- compare the approaches of several existing methods,
- compare appropriateness and performance of the methods on benchmark problems.

The text is organised in the following chapters:

- Chapter 1 introduces the techniques of traditional tree-based GP, the necessary concepts from formal language theory, and several grammar-based GP methods. Common features and issues are pointed out.
- Chapter 2 presents applications that we will use for comparison and benchmarking.
- Chapter 3 describes several experiments with grammar-based methods in the presented applications, and analyses the results.
- Chapter 4 provides information about the implementation used for the experiments, which is available on the accompanying medium and online[1].
- The closing chapter concludes the thesis and suggests possibilities for further research.

[1] `http://nohejl.name/age/`

Chapter 1

Grammar-Based Genetic Programming Methods

In this chapter we will introduce several methods for genetic programming based on formal grammars, assuming basic knowledge about evolutionary algorithms, particularly genetic programming and *genetic algorithms* (GA), and formal grammars. If you are not familiar with evolutionary algorithms, the textbooks by Goldberg (1989) (on genetic algorithms) or Poli et al. (2008) (on genetic programming in a broad sense) provide a good overview. Alternatively, you can find a short summary of the commonest techniques in my bachelor thesis (Nohejl, 2009).

We will begin with an informal review of the basic concepts and methods that preceded grammar-based genetic programming:

- in Section 1.1, we will describe the grammars and notations for them commonly used in grammar-based methods,
- in Section 1.2, we will review the plain tree-based genetic programming and one of its developments highlighting the points that will later interest us,
- in Section 1.3, we will outline how grammars were used to encode bias in inductive logic programming.

Then, we will describe the following grammar-based GP methods:

- in Section 1.4, context-free grammar genetic programming,
- in Section 1.5, LOGENPRO, a genetic programming system based on logic grammars,
- in Section 1.6, grammatical evolution.

To complete our tour we will discuss the common features and shortcomings of the methods in Section 1.7 and examine the implications for their performance in applications in Section 1.8.

1.1 Formal grammars and programming

Context-free grammars (CFGs) are used to express syntax of most currently used programming languages because they provide a reasonable trade-off between expressiveness (relative size of the class of expressible languages) and efficiency, which is especially important for syntactic analysis (parsing) of programs.

This also makes them a natural choice for augmenting genetic programming with grammars. Most of the methods that we are going to discuss are based on CFGs, while the remaining use grammars that extend CFG in some way. We will therefore begin with a precise definition and the corresponding terminology and notation:

Definition. A *context-free grammar*, or *CFG*, is formed by four components (adapted from Hopcroft et al., 2000, ch. 5):

1. There is a finite set of symbols that form the strings of the language being defined. We call this alphabet the *terminals*, or *terminal symbols*.

2. There is a finite set of *nonterminals*, or *nonterminal symbols*.[1] Each nonterminal represents a language; i.e., a set of strings.

3. One of the nonterminals represents the language being defined; it is called the *start symbol*. Other nonterminals represent auxiliary classes of strings that are used to help define the language of the start symbol.

4. There is a finite set of *productions* or *rules* that represents the recursive definition of a language. Each production consists of:

   (a) A nonterminal that is being (partially) defined by the production. This nonterminal is often called the *head* of the production.

   (b) The production symbol $\rightarrow$.

   (c) A string of zero or more terminals and nonterminals. This string, called the *body* of the production, represents one way to form strings in the language of the nonterminal of the head. In so doing, we leave terminals unchanged and substitute for each nonterminal of the body any string that is known to be in the language of that nonterminal.

When applying the rule, we say that it is used to *rewrite* its head to its body, or that using the rule a string containing the body *derives* from the string containing the head. The terminology that we have introduced can be used for other kinds of grammars as well. It is the restriction that production heads consist of a single nonterminal that gives context-free grammars their name by ruling out context dependence.

Formally, a CFG is usually represented as the ordered quadruple $G = (N, T, P, S)$, where N is the set of nonterminals, T the set of terminals, P the set of productions, and S the start symbol.

[1] Hopcroft et al. (2000) prefer the name *variables*, which is reserved for its more common use in this text.

Notation. We will use a number of conventions when working with CFGs and grammars in general:

1. Letters, digits and symbols in upright or non-proportional type represent terminals. Examples: x, 4, ×, %.

2. Identifiers consisting of lower-case letters in italic type represent nonterminals. Examples: *s*, *var*, *expr*.

3. Uppercase letters in italic type are used as meta-variables (symbols that stand for an unspecified nonterminal, or less often terminal). Examples: A, B, X.

4. Lower-case Greek letters stand for strings consisting of terminals and nonterminals, λ denotes an empty string, and period is used to make concatenation explicit. Examples: α, β, $\xi = \xi.\lambda$.

Note particularly the representation of nonterminals, which is contrary to the usual convention of the formal language theory, but allows us to use more expressive identifiers.

We will also use the *Backus-Naur form* (BNF) as a notation for context-free grammars. Examples of a CFG describing simple arithmetic expressions and a corresponding BNF notation are provided in Listing 1.1 and Listing 1.2. See Naur (1963) for a formal definition.

Definition. Let $G = (N, T, P, S)$ be a context-free grammar. The *derivation trees*, or *parse trees*, for G are rooted, ordered trees that satisfy the following conditions (adapted from Hopcroft et al., 2000, ch. 5):

1. Each internal node is labelled by a nonterminal in N.

2. Each leaf is labelled by either a nonterminal, a terminal, or λ. However, if a leaf is labelled λ, then it must be the only child of its parent.

3. If an internal node is labelled A, and its children are labelled $X_1, X_2, \ldots, X_k$ respectively, from the left, then $A \to X_1 X_2 \cdots X_k$ is a production in P.

The *yield* of a derivation tree is the concatenation of its leaves in the order they appear in the tree. The *depth* of a node in a tree is the length of the path from root to that node counted as the number of edges. The *height* of a tree is the largest depth of a node in the tree. Thus root node has depth 0, and the minimum height of a tree whose yield consists of terminals and λ is 1. These definitions are in line with the standard textbook terminology (Hopcroft et al., 2000, ch. 5; Cormen et al., 2001, sec. B.5.2), and can be easily extended to derivation trees for other types of grammars, and in case of depth and height to any rooted trees. (See Figure 1.1 for an example.) There is no formal difference between a derivation tree and a parse tree: the former emphasises the generative aspect, the latter emphasises the aspect of syntactic analysis.

$$
\begin{array}{ll}
\mathit{expr} \to (\,\mathit{expr}\ \mathit{op}\ \mathit{expr}\,) & \mathit{expr} \to \mathit{prim} \\
\mathit{op} \quad \to + & \mathit{op} \quad \to \times \\
\mathit{prim} \to \mathrm{x} & \mathit{prim} \to 1.0
\end{array}
$$

Listing 1.1: Production rules for a CFG whose set of nonterminals is $\{\mathit{expr}, \mathit{op}, \mathit{prim}\}$, its set of terminals is $\{(,), +, \times, \mathrm{x}, 1.0\}$, and its start nonterminal is *expr*.

```
<expr> ::= ( <expr> <op> <expr> ) | <prim>
<op>   ::= + | *
<prim> ::= x | 1.0
```

Listing 1.2: A BNF version of the previous example. Note that | denotes alternatives, and that the sets of terminals and nonterminals are implied, as is the start nonterminal following the convention of putting its productions first.

Let's state informally several basic facts about context-free grammars:

Fact 1. *The same language can be described by multiple context-free grammars.* Consider the rules $s \to p$, $s \to q$, $s \to 1$, $p \to s + s$, $q \to s \times s$, the rules $s \to s + s$, $s \to s \times s$, $s \to 1$, and the rules $s \to s + s$, $s \to s \times s$, $s \to s + s \times s$, $s \to 1$.

Fact 2. *A CFG may be ambiguous.* Consider the rules $s \to s + s$, $s \to s \times s$, and $s \to 1$, and a string $1 + 1 \times 1$. It is impossible to tell if it was derived by first using the first rule or the second rule. Thus, for an ambiguous CFG, different derivation trees can yield the same string.

Fact 3. *A CFG cannot express all what is usually considered part of a programming language syntax.* Notably, the use of declared variables is context-dependent. Consider a language with a "`let` *var* `=` *expr* `in` *expr*" construct: by syntactic analysis (or generation) according to any given context-free grammar, it is impossible to ensure that each *var* nonterminal occurring in the derivation of the second *expr* nonterminal is rewritten to an identifier declared in an enclosing `let` construct.

Fact 4. *For a given CFG the set of strings yielded by parse trees such that (1) they are rooted in the start symbol, and (2) they yield a terminal string, is the language defined by the CFG.* A corresponding derivation of a terminal string from the language can be constructed from any given derivation tree and vice versa.

Definite clause grammars (DCGs) are a formalism tied to the Prolog programming language and related to its early application to natural language processing. Rather than being another type of a formal grammar, they are a specific notation for grammars with semantics derived from logic programming. In the context of logic programming languages such as Prolog or Mercury, they are often used to create complex parsers.

DCGs can easily capture long distance dependencies, and can be used as a natural notation for context-free grammars, as well as for the more expressive attribute grammars (as shown by Sterling and Shapiro, 1994, ch. 19), which are

```
expr --> ['('],expr,op,expr,[')'].    expr --> prim.
op   --> ['+'].                       op   --> ['*'].
prim --> ['x'].                       expr --> ['1.0'].
```

Listing 1.3: A DCG version of the previous example. Note that commas denote concatenation and square brackets enclose terminals, which can be strings, as in the example, or any terms.

```
s           --> rep(N,a), rep(N,b), rep(N,c).
rep(end,_)  --> [].
rep(s(N),X) --> [X], rep(X,N).
```

Listing 1.4: A DCG grammar for $a^n b^n c^n$, a language that cannot be described by a context-free grammar. Note the Prolog variables `X` and `N`, the number of repetitions expressed using the term structure `s(s(···s(end)···))`.

```
expr(V) --> ['let'],X,['='],expr(V),['in'],expr([X|V]),{var(X)}.
expr(V) --> X, {member(X,V)}.
```

Listing 1.5: A fragment of a DCG grammar for checking variable declaration in a "`let` *var* `=` *expr* `in` *expr*" construct. Note the Prolog variables `V`, `X` and the external predicates `var/1`, `member/2` in curly braces.

commonly employed for syntactic analysis in compilers rather than plain CFGs. The looser term *logic grammars* may refer to DCGs or a derived formalism of equal or restricted expressive power (see Section 1.5, Section 1.3).

In the simplest case a DCG expresses a context-free grammar (Listing 1.3). As in CFGs, a head of a production consists of a single item, and its body is a string of items. The items, however, may be arbitrary terms with Prolog variables (Listing 1.4), and further extending the computational power, a list of arbitrary Prolog goals may be added to each rule (Listing 1.5). As shown in Listing 1.4 and Listing 1.5, a DCG can describe context-dependency such as a restriction to use only declared variables in a programming language syntax.

A precise definition and more complex examples can be found in *The Art of Prolog* by Sterling and Shapiro (1994, ch. 19, 24), and Sperberg-McQueen (2004) provides a good practical overview online. The semantics should intuitively be clear to readers familiar with logic programming: the notation is translated into a Prolog program for a top-down left-to-right parser resulting in a clause for each production with any goals in square brackets being added to the body of the clause. When the parser is executed, the terms and variables used in the grammar are subject to unification. Finally, let's state an obvious fact:

Fact 5. *The DCG formalism has enough power to express unrestricted grammars.* Prolog is Turing-complete (Sterling and Shapiro, 1994, sec. 17.2), and any Prolog goals can be added to the grammar rules.

1.2 Genetic programming

In genetic programming, as pioneered by John Koza and described in his book *Genetic Programming* (1992), candidate solutions are programs represented by trees. In Koza's programming language of choice, such representation does not involve any extra costs: when evolving Lisp programs in a Lisp-based system, the program, the corresponding tree, and the data that represents it coincide (see Figure 1.1 for comparison between a GP tree and a derivation tree). In GP terminology, inner nodes of a tree are drawn from a set of *functions*, while its leaves are drawn from a set of *terminals*[2]. Except for quantitative parameters and a fitness function, the two sets are the only input of a genetic programming algorithm, and thus they determine its search space.

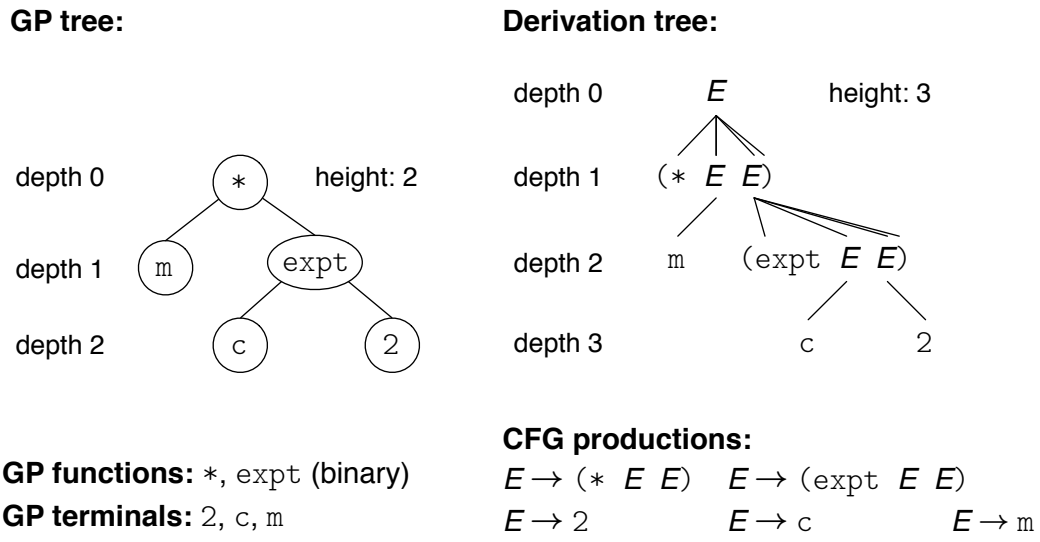


Figure 1.1: An individual tree in GP and the corresponding derivation tree (parse tree) for a CFG, both representing the Lisp expression `(* m (expt c 2))`. In the context of tree-based GP, the formal distinction is often neglected and individual trees are called parse trees (Koza, 1992). The CFG with a single nonterminal expresses the closure property of GP functions and terminals.

For the search to be effective, the sets are required to possess the *closure* property, quoting Koza (1992, sec. 6.1.1): "each of the functions in the function set [must] be able to accept, as its arguments, any value and data type that may possibly be returned by any function in the function set and any value and data type that may possibly be assumed by any terminal in the terminal set." Without this property, invalid programs would often arise from initialisation (by randomly generating trees), and then repeatedly from crossover (by swapping arbitrary subtrees) and mutation (by replacing an arbitrary subtree with a randomly generated tree).

[2] Note the difference in terminology between the terminology of GP and that of formal grammars. When necessary to prevent ambiguity, I will refer to GP terminals and GP functions.

The other required property stated by Koza is *sufficiency*: if the functions and terminals are not sufficient to represent a solution, no solution can be found. This is a general problem of all machine learning algorithms: analogously, inputs and outputs of a neural network need to be assigned before training the network. A more subtle point, however, is worth noting: not all sufficient sets of terminals and nonterminals result in equally efficient searches.

Satisfying the closure property was not intended only as a workaround for the issue of syntactic invalidity but also for runtime errors (for instance division by zero). It proved to be an effective solution in some cases, but it is not feasible for problems that demand extensive use of several mutually incompatible types of values (such as scalars, vectors and matrices of different dimensions). The obvious way to solve this is to restrict the genetic operators ad hoc, as shown by Koza (1992, ch. 19).

Koza (1992) stressed that genetic programming is, like GA (Goldberg, 1989), a weak method: the search algorithm is problem-independent. He also emphasised the positive consequences, universality and ability to "rapidly [search] an unknown search space", over the nontrivial issues of adapting such a method to a specific problem, which we anticipated in the discussion of closure and sufficiency. These issues motivated the development of the more advanced genetic programming techniques that we are going to discuss.

One such early technique, a more general way of restricting the search space than Koza's ad hoc modification of operators, was devised by David Montana (1994). In his *strongly-typed genetic programming* (STGP), all functions and terminals have types, which are used to automatically constrain how trees can be composed and modified. As noted by Montana, although this is a cleaner method, it is in the end equivalent with Koza's solution.

What Montana thought of as a "big new contribution" was the introduction of generics, which allows you to specify for instance a general function for matrix multiplication, which takes an $m \times n$ matrix and an $n \times p$ matrix and returns an $m \times p$ matrix, where m, n, and p are arbitrary integers. The usefulness of generics beyond specific problems with multi-dimensional structures hinges on the assumption that "the key to creating more complex programs is the ability to create intermediate building blocks capable of reuse in multiple contexts," and that generics induce an appropriate level of generality. Whether STGP with generics really results in more reusable building blocks within a single complex problem remains to be confirmed[3] but the idea of restricting the search space of the GP algorithm declaratively by formal rules has since become widespread.

Another important technique that extends in tree-based GP are *automat-*

[3] Arguably, one of the more complex problems to which genetic programming has been applied was evolving a strategy for a virtual soccer team in the RoboCup competition. The algorithm ran for months on a 40-node DEC Alpha cluster to evolve a "team" that won its first two games against hand-coded opponents and received the RoboCup Scientific Challenge Award (Luke et al., 1997; Luke, 1998). While the authors did use STGP, they employed its basic form without generics.

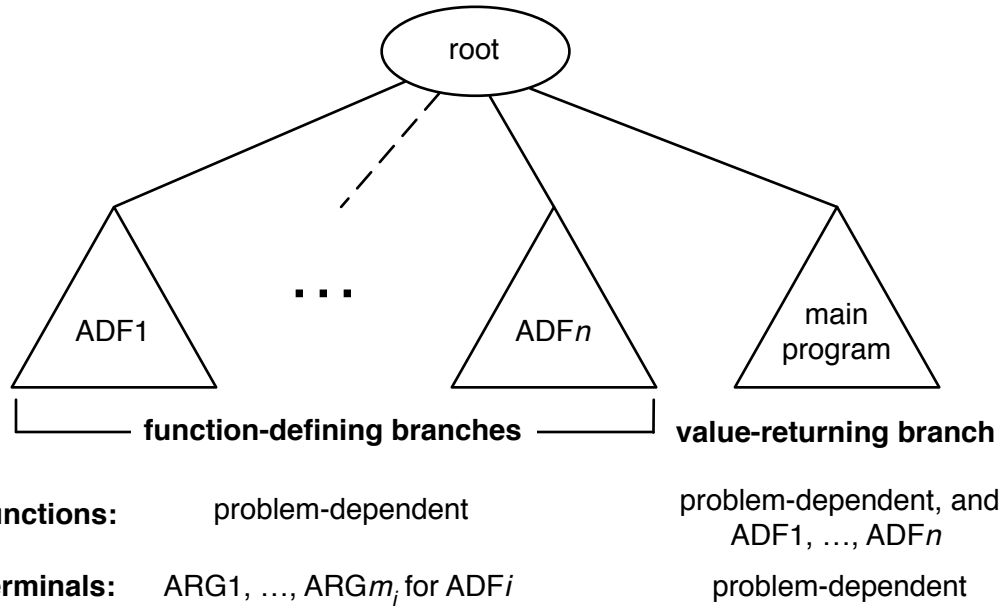


Figure 1.2: ADFs in tree-based GP: the partially constrained structure of individuals' trees for n ADFs, where the ith ADF has m_i parameters.

ically defined functions (ADFs). ADFs, as originally adopted by Koza (1992, ch. 20), provide a simple mechanism for co-evolution of subroutines. Since then a number of more sophisticated variants have been proposed, but we will describe the original version. The idea is inspired by the practise of human programmers: in the same way as man-made programs are usually composed of the main function and additional reusable functions, the individual trees in GP may be composed of a *value-returning branch* and several *function-defining branches*. The number and order of these branches is fixed, as is the number of formal parameters of the ADF defined by each function-defining branch. Based on these numerical parameters, new GP functions `ADF1`, . . . , `ADF`n, which refer to the ordered function-defining branches, and GP terminals `ARG1`, . . . , `ARG`m, which refer to their formal parameters, are added to the problem-dependent sets of GP functions and GP terminals. The new terminals may be used only in function-defining branches, and the new functions may be used only in the value-returning branch.

While the inspiration by procedural programming is obvious, ADFs are not meant to make the resulting programs more human-readable. Instead they are a way to adapt genetic programming to problems in which symmetry or repetition are expected, thus a way to embed a special kind of problem-dependent knowledge in the general GP algorithm. In order to use ADFs, the standard initialisation procedure and genetic operators need to be modified to preserve these constraints common to all individuals (Figure 1.2).

Thus ADFs place further constraints on the search space, while the traditional genetic programming does not provide a general mechanism for satisfying such additional constraints.

1.3 Grammatically biased ILP

In parallel with the beginnings Koza's genetic programming and its first variants such as STGP, grammars already started to be used in *inductive logic programming* (ILP), another branch of machine learning research[4] that emerged at the time. As we will later see, this is where the grammar-based GP methods drew inspiration.

ILP constructs a hypothesis in the form of a logic program, more precisely a Prolog program, from a set of positive examples, from a set of negative examples, and from *background knowledge*, also a Prolog program. The article by Muggleton (1994) provides a concise description of the basic techniques of ILP and the theory behind it. It also acknowledges the importance of problem-dependent knowledge to restrict the search space: "in order to ensure efficiency, it is usually found necessary to employ extra-logical constraints within ILP systems". Two categories of such constraints are discussed: "statistical confirmation" via a *confirmation function*, which "fits a graded preference surface to the hypothesis space", and *language bias*, which "reduce[s] the size of the hypothesis space".

From the point of view of evolutionary algorithms, the confirmation function is simply a type of fitness function. We will focus our attention the other kind of constraint, language bias: particularly interesting is a "generalised" approach that "provides a general purpose 'declarative bias'" (as described by Muggleton, 1994). (In this context, "bias" is used to encompass both restriction and preference.) This approach devised by Cohen (1992, 1994) consists of translating the background knowledge into a grammar in a way that guides the formation of a hypothesis. Cohen noted that various existing ILP methods, each using a different algorithm, were well-suited for different problems, and felt that the search bias embedded in the algorithms should instead be "compiled into" the background knowledge. To achieve this he used "antecedent description grammars", which are a special case of definite clause grammars that retains the use of arbitrary terms and unification among these terms (as shown in the Listing 1.4) but does not allow adding external goals to the productions.[5] Such grammars describe antecedents (bodies) of Prolog clauses of a hypothesis.

To search with a weak bias, the grammar could allow various combinations of problem-specific predicates to occur in a clause body, while a strong bias could prescribe a mostly fixed body and restrict the variation only to its part. Unification is used both to ensure that variables from the head of a clause

[4] Koza (1992) originally considered GP a machine learning parading, similarly, Muggleton (1994), who conceived the original ILP, considered it a machine learning framework. Genetic programming can, however, be applied to search and optimisation problems likely to be considered out of the traditional scope of machine learning.

[5] A notation $A \rightarrow \alpha$ `where` P, where P is a Prolog goal, superficially similar to the curly-bracketed goals in DCGs, is introduced in the 1994 article. It is, however, clarified that the goal P is evaluated with regards to A and α "by a macro-expansion process when the grammar is read in".

are used in its body and to impose further constraints: for instance that two predicates should share a variable, or that variables in some predicate should be of the same type. In addition to the "hard constraint" defined by a grammar, Cohen (1992) uses what he calls "preference bias": some production rules are marked "preferred" and some "deferred". Only if the system does not succeed using the preferred productions, it resorts to the deferred ones.

In his articles, Cohen (1992, 1994) shows how to emulate different strategies, including that of a well-known ILP system FOIL, and how to improve on FOIL's performance by adding various kinds of background knowledge using only antecedent description grammars: "The contribution of this paper is to describe a *single* technique which can make use of *all* of these types of background knowledge—as well as other types of information about the target concept—in a uniform way." As we will show in the next section, the concept of declarative bias and some of these mechanisms can be transposed to genetic programming. In Section 1.5 we will describe a more recent system integrating ILP and GP, which shares even more details with Cohen's methods.

1.4 CFG-GP, language bias, and search bias

The first notable use of formal grammars to control the search algorithm of genetic programming probably came from Peter Whigham as both another, in a way more general, solution to the typing problem recognised by Montana, and a means of introducing more bias into genetic programming (Whigham, 1995, 1996). In the 1995 article Whigham noted that a context-free grammar can be used in similar ways as types to restrict the structure of candidate solutions, in his terminology, to introduce language bias. The proposed method, called *context-free grammar genetic programming* (CFG-GP), is based on a straightforward redefinition of the elements of tree-based GP to respect a given context-free grammar. The individuals still have the form of trees, but instead of representing Lisp expressions, the trees are derived according to an arbitrary CFG, and genetic operators are altered to preserve this representation.

We have already mentioned a coarser form of language bias, which is created by the sets of terminals and functions in original GP, but a CFG allows to embed more problem-dependent knowledge and also to easily use the programming language most appropriate for a given problem.

Whigham (1995) also proposes the following mechanism for learning bias:

- Let each production rule in the grammar have an integer weight (called "fitness" by Whigham, although it's a different concept from individual fitness) and let the weights initially be equal to 1.

- In each generation: Find the fittest individual (choosing one of the least high among equally fit); choose one of the deepest nonterminals B in its derivation tree; let α be the string of terminals into which it is rewritten; if B is a singleton, then let A be the parent of the highest node in the chain

of singletons that ends in B, else let $A = B$. Create a new production rule $A \rightarrow \alpha$; if $A \rightarrow \alpha$ is already in the grammar, increase its weight by 1, otherwise add $A \rightarrow \alpha$ with weight 1 to the grammar.

- When applying mutation or "replacement" (creating new individuals): select production rules for a given nonterminal with probability proportional to their weights.

The additional operator called replacement simply replaces a fixed part of the population with individuals created in the same way as when creating the initial population. The learnt bias thus affects the mutation and "replacement" operators in each generation, and can also be used in a subsequent run of the algorithm.

Later, Whigham (1996) emphasised the distinction between three kinds of bias: selection bias, language bias, and search bias. In his terminology selection bias is the compound effect of a selection scheme and the fitness function, language bias consists of the restriction imposed by language (grammar), and search bias consists of the factors that control search (crossover and mutation). Seen from this perspective, the bias learning in Whigham's original article compiled search bias into language bias. In contrast to this, the 1996 article presents a mechanism to control search bias separately:

- "Selective" versions of the mutation and crossover operators are introduced. A selective operator may be applied only to a subtree rooted in a nonterminal from a particular set. Several instances of such a selective operator may be used, each with a different probability. These probabilities are considered the principal means of search bias. Note that the probabilities do not govern frequencies of particular nonterminals in the population, but frequencies of operator application to them.

- Production rules in the grammar may still be assigned weights (in this article called "merit weighting"), but the weights are only constant, and no new production rules are added to the grammar. The weights apply to initialisation and mutation as already described, and are presumably part of search bias.

Whigham's approach has shown that grammars, in this case CFGs, can be used in GP in a straightforward manner to constrain the search space. As noted by Whigham (1995) without going into detailed comparison, it is a "different approach to the closure problem [than STGP]." A context-free grammar can be used both to express constructs of a wide range of languages and to emulate a type system with a small finite set of types by having one nonterminal for each. STGP with its generics (Montana, 1994) is more powerful in this regard, but such power comes with a performance trade-off (Poli et al., 2008, sec. 6.2.4).

Additionally, Whigham has used grammars a vehicle for a finer control of bias. In the two articles, Whigham (1995, 1996) proposed two different ways of working with bias: in the former using means analogous to those that Cohen

used in ILP (see Section 1.3: knowledge is being compiled into grammar, some rules may be preferred to others), but added a simple learning mechanism; in the latter he abandoned learning and tried to keep search bias separated from language bias, while still taking advantage of using a grammar. Neither of these approaches would be possible if the language bias wasn't specified declaratively.

1.5 Logic grammar based genetic programming: LOGENPRO

Wong and Leung (1995) presented a genetic programming system based on logic grammars, called LOGENPRO (LOgic grammar based GENetic PROgramming system), and later (2000) the same authors published a book on this system. LOGENPRO is presented as "a framework [. . .] that can combine GP and ILP to induce knowledge from databases" (Wong and Leung, 2000). The system is based on the same core algorithm as tree-based GP or CFG-GP: iterated application of fitness-based selection and genetic operators, but instead of context-free grammars, LOGENPRO employs logic grammars "described in a notation similar to that of definite clause grammars".

The only difference other than in notation between DCGs and these logic grammars seems to be that the "logical goals" that can be added to rules in LOGENPRO are not strictly limited to logic goals as used in Prolog: they are in fact procedures defined in Lisp, the language in which the framework itself is implemented. (See Listing 3.3 on page 47 for an example of a LOGENPRO grammar.) The framework emulates the mechanisms of logic programming to interpret the grammar, but it does not feature a complete or cleanly separated logic programming environment.[6]

LOGENPRO does not use any mechanisms or algorithms specific to ILP but Wong and Leung (2000) demonstrate that with a suitable grammar, it can be used to learn logic programs and it achieves results competitive with earlier ILP systems not based on GP and grammars. What differentiates it from CFG-GP is the more powerful formalism for grammars, which is closer to the one used by Cohen in ILP. As we have noted in Section 1.1, definite clause grammars can be used to describe the context-dependent constructs often found in programming languages. The representation of individuals in LOGENPRO is still conceptually the same as in CFG-GP (a derivation tree, although with structured nodes, as they can contain terms and goals), but operators need to be much more complex to respect the grammar.

While DCGs are essentially Turing-complete (see Fact 5, page 7), LOGENPRO does not evaluate logical goals except when generating new trees:

[6] I obtained LOGENPRO source code from the authors via personal communication. Although Wong and Leung (1995, 2000) present the "logic grammars" used in LOGENPRO as different from DCGs, no difference is evident from their description, and no details about the implementation are given.

the subtrees that contain logic goals cannot be changed by the operators (Wong and Leung, 2000, sec. 5.3), so they behave as atomic and immutable components throughout the evolution (compare with the approach used by Cohen, 1994, see footnote 5, page 11).

On one hand, even the remaining power of DCGs, which lies in the use of variables and unification, still make operators, particularly crossover, quite complex: according to Wong and Leung (2000, sec. 5.3), the worst-case time complexity of crossover is $O(m \cdot n \cdot \log m)$, where m and n are the sizes of the two parental trees, slightly higher than $O(mn)$ in Koza's GP with ADFs and Montana's STGP. On the other hand, the same power allows it to emulate the effect of both of these methods (Wong and Leung, 2000). The scheme of individuals that use ADFs (as shown in Figure 1.2) can be easily embedded in a grammar (see Section 3.7, also demonstrated by Wong and Leung, 2000). While the authors do not explain how exactly LOGENPRO can emulate STGP including generics, a finite set of types can be emulated even using a CFG (as we have remarked in Section 1.4).

LOGENPRO focuses on sophisticated constraints on the search space that can be described declaratively using a DCG, and performs a search using elaborate operators that preserve these constraints. It does not provide any special mechanisms for learning bias, or any additional parameters for the search.

1.6 Grammatical evolution

Grammatical evolution (GE) (Ryan et al., 1998; O'Neill and Ryan, 2003) is a recent method for grammar-based GP that, unlike LOGENPRO or CFG-GP, significantly changes the paradigm of traditional tree-based GP by introduces the notion of *genotype-phenotype mapping*. In a parallel to the biological process of gene expression, each individual has a variable-length linear *genotype* consisting of integer codons, to which the genetic operators such as mutation and crossover are applied. In order to evaluate the individual's fitness, the genotype is mapped to *phenotype*, a program in the language specified by the given context-free grammar. Trees are not used for individual representation but implicitly as a temporary structure used in the course of mapping.

The mapping used in GE is so simple that we can describe it in full details (adapted from Nohejl, 2009):

> In analogy to the DNA helix and nucleobase triplets, the string is often called *chromosome*, and the values it consists of are called *codons*. Codons consist of a fixed number of bits. The mapping to phenotype, proceeds by deriving a string as outlined below in the pseudocode for Derivation-Using-Codons. The procedure accepts the following parameters:

– G, a context-free grammar in BNF. Note that BNF ensures that there is a non-empty ordered list of rewriting rules for each nonterminal.

– S, a nonterminal to start deriving from. The start nonterminal of G should be passed in the initial call.

– C, a string of codons to be mapped. Note that it is passed by reference and the recursive calls will sequentially consume its codons using the procedure Pop.

```
Derivation-Using-Codons(G, S, C)
 1  P ← array of rules for the nonterminal S in G indexed from 0
 2  n ← length[P]
 3  if n = 1                          ▷ only one rule (no choice necessary)
 4     then r ← 0
 5  elseif length[C] > 0              ▷ choice necessary, enough codons
 6     then r ← Pop(C) mod n
 7  else error "Out of codons"        ▷ or wrap C, more on that later
 8  σ ← body of the rule P[r]
 9  τ ← λ
10  foreach A ← symbols of σ sequentially
11        do if A is terminal
12              then τ ← τ . A
13              else  τ ← τ . Derivation-Using-Codons(G, A, C)
14  return τ
```

Out of the context of fitness evaluation, the individuals are simple binary strings to which standard GA (Goldberg, 1989) operators for one-point crossover and mutation are applied (the only differences are that the chromosomes are variable-length and the crossover is applied on codon boundary, not at arbitrary position), the population can also be initialised by generating random binary strings. Thus the operators for GE can be implemented very efficiently, and the performance penalty of performing the mapping is also low.

It would appear that this efficiency comes at the cost of a high proportion of invalid individuals (see line 7 of the pseudocode). This issue is addressed by *wrapping* the chromosome (interpreting it in a circular fashion) if needed, which can greatly reduce the number of invalid individuals (O'Neill and Ryan, 2003, sec. 6.2). Still, the operators clearly may have a different effect than the traditional GP operators, which tend to preserve most of the individuals' structure and are designed to have a predictable effect. In GE, on the one hand, minor changes in genotype can translate into massive changes in phenotype; on the other hand, some parts of genotype (and thus any changes to them) may not have any effect on the phenotype. O'Neill and Ryan (2003) justify these issues as parallels to genetic phenomena (such as *genetic code degeneracy* in the case of unused genetic information), and show that in some cases they can improve performance.

While a formal analysis of the general effect of these simple operators on phenotype, and thus the overall search performance, is lacking, the effect can be measured and compared statistically in specific applications. This method is used by O'Neill and Ryan (2003) to compare different variants of operators, and we will use it to compare performance with CFG-GP in Chapter 3.

Compared with CFG-GP, and especially with LOGENPRO, grammatical evolution has an extremely simple implementation that consists of highly efficient elements. We could call these elements (operators, simple random initialisation) grammar-agnostic: they do not depend on the grammar at all, and the search bias that they create cannot be adjusted with regards to the grammar. The constraints of the grammar are ensured by the genotype-phenotype mapping at the expense of preservation of phenotypic structure that the traditional GP, as well as CFG-GP or LOGENPRO, strive for.

1.7 Common features and shortcomings

Figure 1.3 provides an overview of the grammar-based GP methods that we have presented along with traditional GP and grammatically biased ILP. The most important trait of all grammar-based methods is that they provide a general framework for declaratively describing the search space that the traditional GP or ILP lacks. One aspect that differentiates them is the power of this declarative description. The relatively weak context-free grammars can describe the basic structure of common programming languages or their subsets (except context-dependent constructs), and emulate simple type systems, and ADFs. The logic grammars (antecedent description grammars used in Cohen's ILP or DCG-like logic grammars in LOGENPRO) can capture context-dependency using logic variables and unification, but their ability to use arbitrary logic goals is of limited use in genetic programming (as exemplified by genetic operators in LOGENPRO, Section 1.5).

The expressive power of grammars entails a performance trade-off for structure-preserving operators. Grammatical evolution seems to avoid this issue by using grammar-agnostic, possibly destructive operators similar to those traditionally used in GA.

Apart from the hard constraints of the search space, the grammar may serve as a vehicle for further bias (preference). This direction was explored by Whigham (1995, 1996) through selective operators and learning of production weights. It may be less obvious that the grammar itself creates a bias by interaction of its form (equal grammars may differ in form, see Fact 1, page 6) and the operators, or the genotype-phenotype mapping in case of GE. The grammar-based methods seem to be designed as if using a particular form of the grammar was an obvious way to embed problem-dependent knowledge, but with their growing complexity (logic grammars in LOGENPRO, interaction of operators and the mapping in GE) this is not the case. (The grammar for the artificial ant problem in Section 3.5, and the grammars for timetabling heuristics

in Section 3.9 will later be discussed in this context.) The otherwise relatively simple CFG-GP provides a mechanism (Whigham, 1995) to adjust the form of the grammar over the course of running the algorithm, but the more recent methods (GE and LOGENPRO) do not address this issue.

1.8 Implications for performance

LOGENPRO seems to be very different from the two other methods by using a more powerful formalism for grammars. This could be a double-edged sword if it was actually used in some application. We will attempt to replicate two experiments done by Wong and Leung (2000) with LOGENPRO (in Section 3.6 and Section 3.7). We will show that the experiments do not actually require a logic grammar, and that the results are comparable with those of CFG-GP.

Grammatical evolution ensures constraints given by the grammar by its genotype-phenotype mapping but its operators can be applied to any part of its genotype regardless of the grammar and the phenotype. This makes any grammatical constraints work in an essentially different way in GE than in CFG-GP or LOGENPRO. We can expect this to cause GE to produce different shapes of trees than the other two methods, which work with derivation trees (considered to represent the phenotype for GE). This in turn will result in a different search space. We will attempt to analyse this effect and link it to differences in performance using experiments with several different applications in Chapter 3.

Techniques for reuse of building blocks are crucial for performance in applications that use such blocks. As we will show, it is easy to carry over the techniques used by Koza (1992) for ADFs to any of the grammar-based methods. In fact, it is far easier to specify the ADFs using a grammar than to add the necessary ad hoc constraints to tree-based GP operators. The same grammar will, however, have a different effect in GE than in the other two methods because operators are applied regardless of the grammar. We will see how this impacts performance in experiments in Section 3.7 and Section 3.8.

	Tree-based GP	Cohen's ILP	CFG-GP	LOGENPRO	GE
Representation	tree (LISP S-expressions)	logic program	tree (derivation tree)	tree (derivation tree)	variable-length integer string (mapped to derivation tree)
Language specification	set of terminals, set of functions with closure property	antecedent description grammar (DCG without arbitrary goals)	context-free grammar	logic grammars with arbitrary goals (analogous to DCG)	context-free grammar
Search guided by	fitness function	positive and negative examples (confirmation function)	fitness function	fitness function	fitness function
Search type	evolutionary algorithm with GP operators	FOIL-based ILP algorithm	evolutionary algorithm with GP-like operators	evolutionary algorithm with GP-like operators	evolutionary algorithm with GE-like operators and genotype-phenotype mapping
Additional search bias	operator probabilities, other EA parameters	–	operator probabilities, other EA parameters	operator probabilities, other EA parameters	operator probabilities, other EA parameters, codon size, initial/ maximum chromosome lengths, wrapping
Additional search bias using grammar	N/A	preferred/deferred productions	nonterminal sets for selective operators; weighted productions	–	–
Additional features	–	macros applied when reading the grammar	possibility of bias learning	application to machine learning, data mining,	genetic code degeneracy

Figure 1.3: Comparison of grammar-based GP methods along with traditional GP and grammatically biased ILP.

Chapter 2

Existing Applications

Genetic programming is a very general method that has been applied to a variety of problem domains from circuit design to the arts (Poli et al., 2008, ch. 12). While methods that are more efficient and have gained more widespread industrial use exist in most of these fields, the strength of GP lies in its applicability to problems in which the precise form of the solution is unknown and analytical methods cannot be employed or would be too resource-intensive.

As we have explained in the previous chapter, the addition of grammars to genetic programming serves primarily as a means of adapting the general method to a particular problem or an implementation language: a grammar can be used both to embed problem-dependent knowledge such as variable types and structural restrictions, and to specify a subset of a programming language (or some other formal language) that we want to use for implementation.

Our goal is to compare the results both among different methods and with previously published results. Thus we will focus on relatively simple applications suitable for comparing the three main methods that we have presented: CFG-GP, GE, and in two instances also LOGENPRO. All applications presented in this chapter have been described in existing literature in conjunction with one of the methods or with traditional GP, usually in order to highlight advantages of these methods. One application, the use of grammar-based GP methods for hyper-heuristic (Section 2.5), stands out a little: as we will see, this area is relatively open-ended and demanding on implementation but also capable of producing heuristics competitive with those designed by humans.

Before we proceed to evaluating the methods in the next chapter, we will describe the chosen applications.

2.1 Simple symbolic regression

Regression is concerned with finding a function that best fits known data. The problem has most commonly been reduced *parametric regression*: finding parameters for a function whose form is specified in advance (e.g., an nth-degree polynomial). Assuming that the chosen form is adequate, such methods may be very efficient. In situations in which the adequate form is not known, ge-

netic programming can be used to solve the more general problem as *symbolic regression*: search for a function represented in symbolic form, as an expression using some predefined operations.

A simple problem of real-valued symbolic regression was used as one of the introductory examples by Koza (1992) and continues to be used both as a basic benchmark for newer GP techniques and in more advanced research applications (Poli et al., 2008, sec. 12.2). When used as a benchmark, as opposed to a real-world application, the target values are precomputed using the known target function, which may be a polynomial. The point is that GP is able to find the expression representing the polynomial in the space of expressions that also involve operations unnecessary for the target function such as division or logarithm.

Fitness of candidate solutions is usually measured as a sum of absolute or squared errors at the given points to which various scaling methods can be applied.

Except for specifying a custom language instead of Lisp, there is little advantage in using grammar-based methods over using tree-based GP for simple instances of the problem. But these simple instances can serve as a test bed for unusual operators, such as those employed in GE, which do not behave analogously to those used in tree-based GP.

More intricate cases of symbolic regression will be presented in Section 2.3 and Section 2.4.

2.2 Artificial ant trail

The artificial ant trail is another classic introductory GP problem. The goal is to navigate an artificial ant so that it finds all food lying on a toroidal floor within a time limit. The pieces of food are placed on a rectangular grid to form a trail with increasingly demanding gaps and turns. The ant can use the actions

– Left: turn left by 90 ° without moving,
– Right: turn right by 90 ° without moving,
– Move: advance one square in the direction the ant is facing, eating any food on that square,

combined using conditional branching on Food-Ahead: test the square the ant is facing for food. All actions take one unit of time, the test is instant.

The problem was originally designed to test evolution of finite-state machines using GA and various trails have appeared in subsequent versions of the experiment. The most common one, also used by Koza (1992) to demonstrate the competence of GP in solving this problem, is the so-called Santa Fe ant trail. When using GP, the solution has a form of a short program, which is executed in a loop until time runs out. The syntactic restrictions placed on the program play an important role that we will discuss when evaluating the experiments. Fitness of candidate solution is measured as the number of food pieces eaten within the time limit.

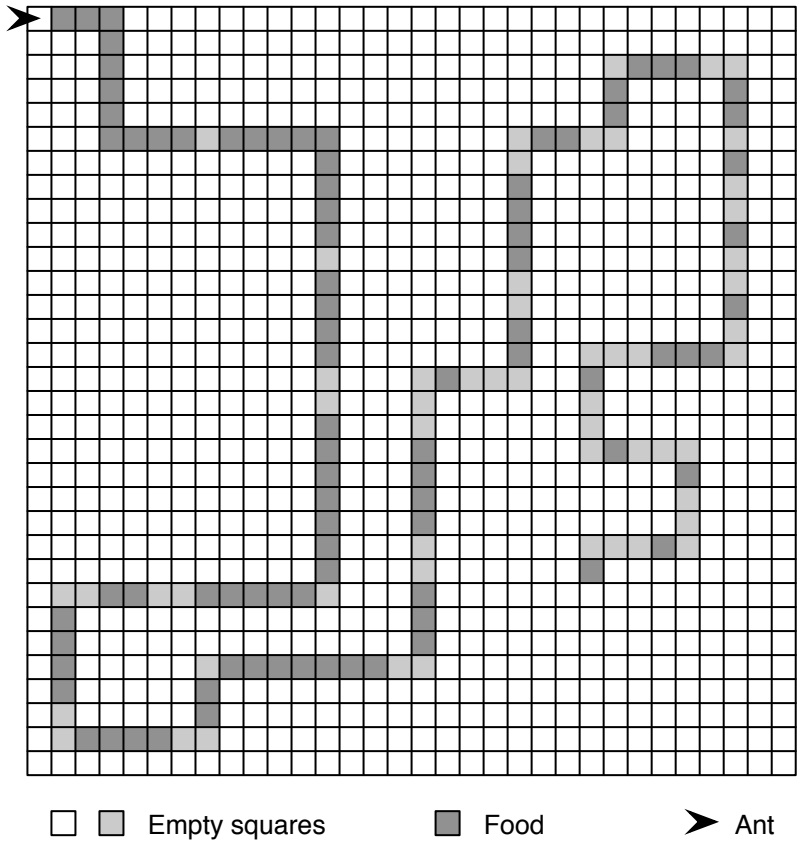


Figure 2.1: The Santa Fe ant trail. 89 pieces of ant food on a 32 by 32 toroidal grid. Ant starts in the north-east corner facing east

2.3 Symbolic regression with multiple types

Symbolic regression can feature values of multiple types: for instance when the independent variables are vectors and the dependent variable is a scalar. It is then desirable to restrict the search space to expressions with vector variables that yield a scalar by applying correct operations to both scalar and vector values. Such search space does not, of course, possess the closure property required by traditional GP. The need to overcome this problem motivated strongly-typed genetic programming (discussed in Section 1.2), but it can be shown that grammars can describe such simple type systems very efficiently.

Wong and Leung (2000) use two such problems to demonstrate that their framework for logic-grammar based genetic programming LOGENPRO can outperform GP when type constraints are involved and that it can also be used to emulate ADFs (briefly introduced in Section 1.2):

1. finding a real-valued function of two three-component vector variables $\vec{x}$, $\vec{y}$ that computes the dot product $\vec{x} \cdot \vec{y}$ (the standard Euclidean inner product, also called scalar product),

2. finding a real-valued function of three three-component vector variables $\vec{x}$, $\vec{y}$, $\vec{z}$ that yields the scalar $\vec{x} \cdot \vec{y} + \vec{y} \cdot \vec{z}$, where $\cdot$ is the dot product, using ADFs.

While Wong and Leung (2000) compare the performance of LOGENPRO with a GP setup that does not use any syntactic constraints, I would suggest that the point should not be that GP cannot use ad hoc syntactic constraints to the same effect but that doing so using a grammar is more general but also far easier and less error-prone: note that even in this seemingly simple example there are three "natural" types of unary and binary operations (scalar-to-scalar, vector-to-vector, vector-to-scalar) and perhaps scalar multiplication.

We will discuss the approach used by Wong and Leung (2000) and demonstrate that other grammar-based methods can be used with similar results. We will also analyse the role of ADFs in the second variant of the problem, which Wong and Leung (2000) did not do properly.

2.4 Boolean symbolic regression

The spaces of k-ary Boolean functions are relatively small (2^{2^k}), as are the sufficient sets of basic functions from which they can be composed (e.g. the set of common logical connectives $\{and, or, not\}$), yet the search space of their symbolic representations is vast. Although symbolic regression of Boolean functions is of no practical interest, these properties make it suitable for evaluating genetic programming systematically. Koza (1992) has demonstrated that the difficulty of finding different ternary Boolean functions by blind random search among expressions of certain maximum size varies considerably and that GP is statistically more successful than blind random search in finding those that are particularly hard to find. (More details in Koza, 1992, ch. 9.)

His experiments have also shown that the hardest to find among ternary functions are the parity functions (the even and odd parity function are true if and only if the number of true arguments is even and odd, respectively). Consequently, the parity functions are relatively hard to find even in the classes of higher-arity Boolean functions. At the same time they can be easily composed from simple building blocks: the exclusive-or functions, and a negation in case of even parity. This in turn makes Boolean symbolic regression of the parity functions suitable for testing the impact of ADFs on performance.

We will use the even-parity Boolean regression problem to test the performance of techniques for the grammar-based methods that emulate ADFs. Results will be compared to those presented by Koza (1992) for tree-based GP with and without ADFs.

2.5 Hyper-heuristics

In contrast to the previously presented applications, *hyper-heuristics* is actually a whole field covering very diverse problems. A hyper-heuristic search engine does not search for problem solutions but rather for heuristics for solving a problem. Thus if the class of problems of interest was Boolean satisfiability,

heuristics for a SAT solver would be sought rather than truth-value assignments.

Different approaches can be used to find heuristics, but if some heuristics have already been developed and can be combined in more complex ways then just choosing a sequence, grammar-based GP can be used to great advantage. The grammar allows us both to use the language of our problem-solving framework directly and to embed our knowledge about the existing heuristics that are to be used as building blocks.

Bader-El-Den et al. (2009) have recently successfully applied a grammar-based GP hyper-heuristic framework to timetabling. The method they used was essentially CFG-GP. We will try to replicate their results with our implementation of CFG-GP, and compare them to the results obtained using grammatical evolution in Section 3.9.

2.6 Other applications

While we will not experiment with other problems, several other areas of application are worth at least mentioning:

The authors of LOGENPRO have developed their system specifically to target data mining problems. They see their logic grammar based framework (covered in Section 1.5) as a combination of two important approaches to data mining: ILP and GP (Section 1.2 and Section 1.3). In addition to artificial problems such as the dot product problem (see Section 2.3 above), they apply LOGENPRO to two benchmark data mining problems (credit card screening using decision trees and the chess endgame problem using logic programming), and to data mining from "real-life medical databases". LOGENPRO provides results competent with other learning systems in the first benchmark and outperforms ILP systems FOIL, BEAM-FOIL, and mFOIL significantly at most noise levels in the latter benchmark. It is interesting to note that in none of the applications presented by Wong and Leung (2000) the specific feature of their system, the power of logic grammars, is used. In particular, their applications use only simple logic goals that could be replaced by short enumerations, and none of the grammars feature context-dependence via unification. Conceivably, any of their results could be replicated using grammar-based methods other than LOGENPRO, in the same way as we will show on the example of the dot product problem in Section 3.6. Thus, grammar-based methods can also be used as a viable data mining framework.

Grammatical evolution has recently been used in dynamic applications, where the grammar itself is described by a (meta-)grammar and co-evolves along with the individual, allowing for further adaptability (Dempsey et al., 2009).

Natural Computing Research & Applications Group at University College Dublin (UCD NCRA), where most of the current work on grammatical evolution is being done, has also used GE to evolve a logo for the group interactively

(O'Neill and Brabazon, 2008). Their web site[1] showcases other unconventional and creative applications including evolution of elevator music. Results in such applications cannot of course be quantitatively measured but they demonstrate the variety of objects and processes that can be described using a grammar and consequently evolved using grammar-based GP methods.

[1] http://ncra.ucd.ie/

Chapter 3

Experiments with Grammar-Based Methods

In this chapter we will perform several experiments with instances of the problems chosen in Chapter 2.

When evaluating CFG-GP and GE, we will use the AGE (Algorithms for Grammar-based Evolution, originally Algorithms for Grammatical Evolution) framework, which is a competent and well-documented implementation of GE and common evolutionary algorithm elements, as I have shown in my bachelor thesis (Nohejl, 2009). For the purpose of these experiments AGE has been extended with CFG-GP algorithm elements implemented according to Whigham (1995). This will allow us to test CFG-GP and GE in almost identical setups. See Chapter 4 for more information about AGE and its implementation of CFG-GP. The software is available on the accompanying medium and online[1].

The performance of LOGENPRO will be evaluated using the implementation that Dr Man Leung Wong, a co-author of the method (Wong and Leung, 1995, 2000), has kindly provided to me.

3.1 Observed characteristics

As a measure of success I adopt the cumulative success rate over a number of runs, as usual when evaluating evolutionary algorithms. The success in each experiment is defined by the *Success predicate* entry in its table (see Section 3.2 below). For the purpose of these experiments, the algorithms are left running until the maximum number of generations is reached even when the success predicate has been satisfied.

To further facilitate comparison, we will use several characteristics of derivation trees applicable to both methods: tree height (defined in Section 1.1), number of choice nodes, and bushiness (both defined below). In the following definitions, let $G = (N, T, P, S)$ be a context-free grammar and ρ a derivation tree for the grammar G.

[1] `http://nohejl.name/age/`

Definition. We will say that a particular node of ρ is a *choice node* if it is labelled with a nonterminal for which P contains more than one production. (Thus a choice of production had to be made at this node.) *Internal choice node* is a choice node that has at least one choice node among its descendants.

When using GE, the number of choice nodes coincides with the number of codons used during the mapping process. Additionally, the number of choice nodes reflects the amount of information contained in a given derivation tree more accurately than the total number of nodes.

Definition. Let n be the number of all choice nodes in ρ, and k the number of internal choice nodes in ρ. The ratio $\frac{n}{k+1}$ expresses the *bushiness* of ρ.

Bushiness is simply the ratio of the total number of choice nodes and the number of their "parents". Note that in any derivation tree that contains at least one choice node, there is a non-empty group of topmost choice nodes (the group has no choice node ancestor): the additive constant 1 in the formula acts as a virtual parent of these choice nodes. The bushiness is thus analogous to the average branching factor, except that it is defined only based on choice nodes.

AGE can report tree characteristics for both GE and CFG-GP. LOGENPRO offers only basic data about fitness and success.

3.2 Setup

Both AGE, and LOGENPRO allow to configure parameters such as selection method, operator probabilities, or maximum tree heights. We will present these parameters for each experiment in a table derived from the "tableau" format used by Koza (1992), and in a modified form by O'Neill and Ryan (2003). In addition to the entries used by either of them, I also specify the details of the algorithm in the entries *Algorithm*, *Selection*, *Initialisation*, *Operators*, and *Parameters* (for other method-specific parameters).

In order to make comparisons statistically relevant, we will use data from at least 100 runs. In the case of AGE, these are always runs from one sequence with random number generator (RNG) seed value 42; in the case of LOGENPRO these are runs with RNG seeds from the sequence $1, 2, 3, \ldots, n$. (LOGENPRO normally seeds the generator with the current time. I have modified its source code to use a fixed number to seed the generator.)

3.3 Statistics

We will use several statistics: *average* using arithmetic mean, *variance*, and *coefficient of variance* (CV), which is a relative measure of dispersion defined as the ratio of the standard deviation to the absolute value of the mean. The statistics will be computed from a population of individuals from a particular generation. When evaluating results from multiple runs of the same setup

we will average these statistics (using arithmetic mean) over the performed runs. If a population contains invalid individuals, they are excluded from the statistics.

When I describe a difference between two sets of numerical results (or rates of success in two sets) as *significant*, it means that the statistical hypothesis that the two sets are samples of a statistical population with the same mean (or that the probabilities of success in the two sets are the same), has to be rejected on the level of statistical significance of 5 %. Accordingly, whenever a difference is said to be *insignificant*, the same hypothesis cannot be rejected on the level of 5 %. Unless stated otherwise, any differences I point out are significant. I use Student's t-test for testing equivalence of means, and a simple test of equal proportions for testing the probabilities of success.[2]

3.4 Simple symbolic regression

We will compare the performance of CFG-GP and GE in a simple instance of the symbolic regression problem (Section 2.1).

3.4.1 Experimental setups 1

The first setup of GE is deliberately based on the parameters used in the symbolic regression example supplied with GEVA[3]. I have used the same setup (except for maximum tree height, see explanation below) in my bachelor thesis (Nohejl, 2009) to show that the codon size higher than 8 bits does not provide substantial advantage, and that AGE achieves a success rate of 706 and 701 out of 1000 runs (with 31 and 8 bits per codon, respectively), which was significantly better than the rate achieved by GEVA. Thus we will also use 8-bit codons in the current experiment.

When possible, we will use the same settings for CFG-GP: elite size, fitness measure, selection scheme, etc., and also the crossover rate will be the same: even though each method has a different crossover operator, the crossover rate of 0.9 is a standard value of crossover probability in GP, which is used systematically by both Whigham (1995, 1996) and O'Neill and Ryan (2003).

The mutation rate for CFG-GP will be, somewhat arbitrarily, set to 0.1. The higher nominal value is meant to reflect that the mutation rate in CFG-GP is equivalent to its effective per-individual mutation rate, whereas the effective per-individual rate in GE is much higher then the 0.02 per-codon rate, and is dependent on the number of used codons and the distribution of nonterminals in each individual. Therefore, no fixed per-individual mutation

[2] Both are computed using the *R stats package* (functions `t.test` and `prop.test`), which is part of the open-source *R Project for Statistical Computing* available at `http://www.r-project.org/`.

[3] GEVA is "an open source implementation of Grammatical Evolution [...], which provides a search engine framework in addition to a simple GUI and the genotype-phenotype mapper of GE." (O'Neill et al., 2008). It is being developed at Natural Computing Research & Applications Group (NCRA) at University College Dublin: `http://ncra.ucd.ie/Site/GEVA.html`.

rate corresponds to a given per-codon rate used in GE. Whigham (1995, 1996) has used mutation rates of 0.15 and 0, respectively, so 0.1 is a conservative choice.

Because CFG-GP uses a maximum tree height for its operators, while GE usually ensures reasonable tree size only implicitly, we will use 20 as maximum tree height for both methods, a limit high enough to have only marginal impact on the results of the GE setup.

The only parameter left is the maximum height of derivation trees created by the CFG-GP initialisation method. We would like to operate on individuals of roughly the same size as in the GE setup, so that both methods are evaluated on search spaces of comparable size. I have experimentally determined that maximum height 5 results in derivation trees of height closest to those generated by the GE setup (as shown in Figure 3.1).

The setup for both methods is detailed in Table 3.1.

Objective:	Find a function of one variable x represented by an expression to fit a given set of the target function values at specified points. The target function is $x^4 + x^3 + x^2 + x$.
Terminal operands:	x, 1.0.
Terminal operators:	$+, -, \cdot$ (all binary).
Grammar:	See Listing 3.1.
Fitness cases:	20 fixed points $-1.0, -0.9, \ldots, 1.9$.
Raw fitness:	The sum of squared errors taken over the 20 fitness cases.
Scaled fitness:	Same as raw fitness.
Algorithm:	Simple with elite size: 10, generations: 101, population: 100.
Selection:	Tournament, size: 3.
GE initialisation:	Random codon string initialisation, length: 200.
CFG-GP initialisation:	"Grow" method (Whigham, 1995), maximum height: 5.
GE operators:	Fixed-length one-point crossover, probability: 0.9. Codon-level mutation, probability: 0.02.
CFG-GP operators:	Crossover (Whigham, 1995), probability: 0.9. Mutation (Whigham, 1995), probability: 0.1.
Common parameters:	Maximum tree height: 20.
GE parameters:	Maximum wraps: 3. Codon size: 8.
Success predicate:	Raw fitness lower than 0.00001 (to allow for floating-point round-off error.)

Table 3.1: Simple symbolic regression, parameters for setups 1, *CFG-GP (1)* and *GE (1)*.

```
<expr> ::= ( <expr> <op> <expr> ) | <var>
<op>   ::= + | - | *
<var>  ::= x | 1.0
```

Listing 3.1: Simple grammar for symbolic regression in Lua.

3.4.2 Results from setups 1

The final success rate of 999 out of 1000 runs achieved by CFG-GP has been substantially higher than the 733 out of 1000 runs achieved by GE, moreover CFG-GP was more successful in all generations and converged very fast (see Figure 3.1). While derivation tree heights suggest that both methods were searching a similar space, a closer look at the tree characteristics (Figure 3.3) reveals differences:

(1) During the first five generations, GE has a very high CV of both tree height and bushiness compared to CFG-GP, and consequently also a high variance of the number of choice nodes. This can be explained by the initialisation method, and does not seem to have an important lasting effect.

(2) After (1), CFG-GP has a significantly larger CV of tree height, and neither method continuously dominates in average height.

(3) CFG-GP produces significantly bushier trees throughout the run. After (1), neither method continuously dominates in CV of bushiness.

(4) The compound effect of (2) and (3) is a higher variance of the number of choice nodes in trees produced by CFG-GP. We can interpret this variance (which is an absolute measure as opposed to CV) in conjunction with the average as the extent of the search space. Note that this is observable even if we do not take into account the portion of CFG-GP results after CFG-GP reaches almost full success rate.

To examine the possibility that grammatical evolution produces less diverse candidate solutions because of inappropriate operator parameters, let's have a look at fitness statistics in Figure 3.2: GE soon reaches higher coefficient of variance than CFG-GP, and the CV keeps growing while average fitness stays nearly constant in GE. Note that the difference is not due to invalid individuals, which are excluded from the statistics.

This shows that the parameters of the operators result in enough phenotypic diversity but fail to produce a favourable distribution of tree characteristics. Additionally, even if a higher mutation rate had the effect of producing better tree characteristics, it would effectively turn GE into a blind search: both its standard operators can be very disruptive (their effect is not localised to a subtree: see the "ripple" crossover discussed in O'Neill and Ryan, 2003, and the codon and bit-level mutation in my bachelor thesis, Nohejl, 2009). The success rate reached in this experiment depends on the high elitism, which compensates for the disruptive effect of operators.

We can conclude that CFG-GP has searched a larger search space (bushier trees of more varied height) and has achieved substantially better results than GE. (Note that the larger search space is not necessarily a superset of the smaller one as GE may explore less bushy trees better). This may not be a single direct cause of the higher success rate but being able to find a solution more reliably while searching through more diverse candidate solutions is certainly beneficial.

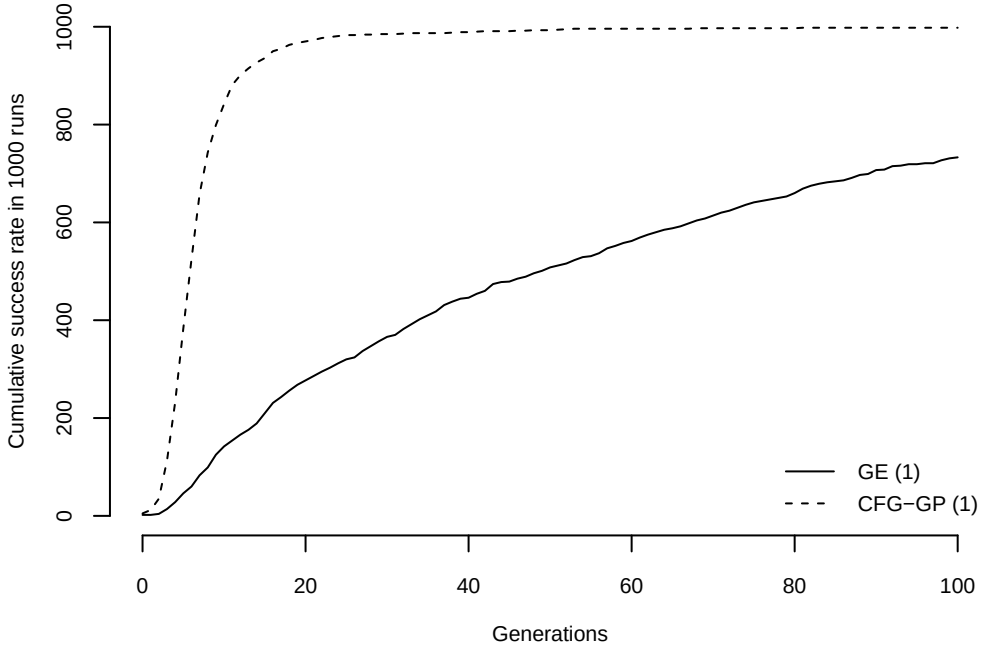


Figure 3.1: Cumulative frequency of success of GE and CFG in symbolic regression of the polynomial $x^4 + x^3 + x^2 + x$, setups 1.

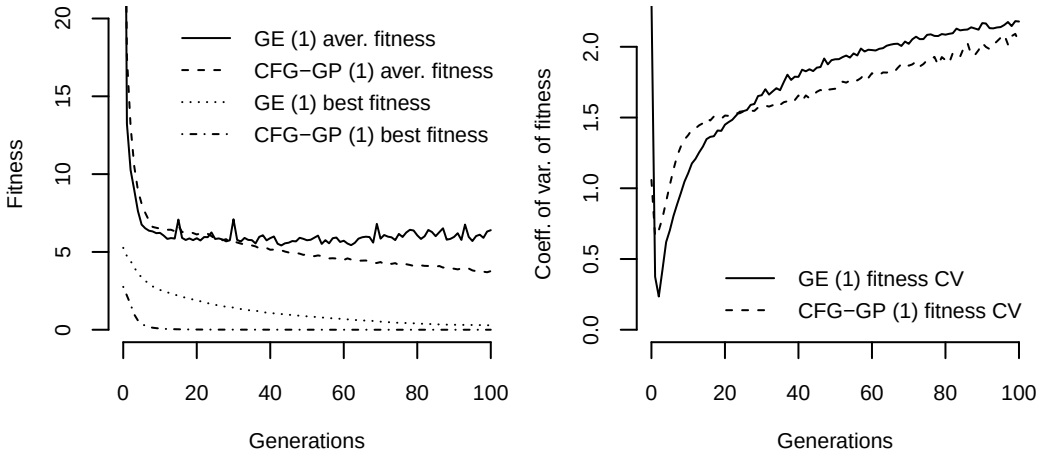


Figure 3.2: Fitness in GE and CFG-GP in symbolic regression of the polynomial $x^4 + x^3 + x^2 + x$, setups 1. The plots show averages, taken over 1000 runs, of the following population statistics: averages, minima (best values), and coefficients of variance of fitness.

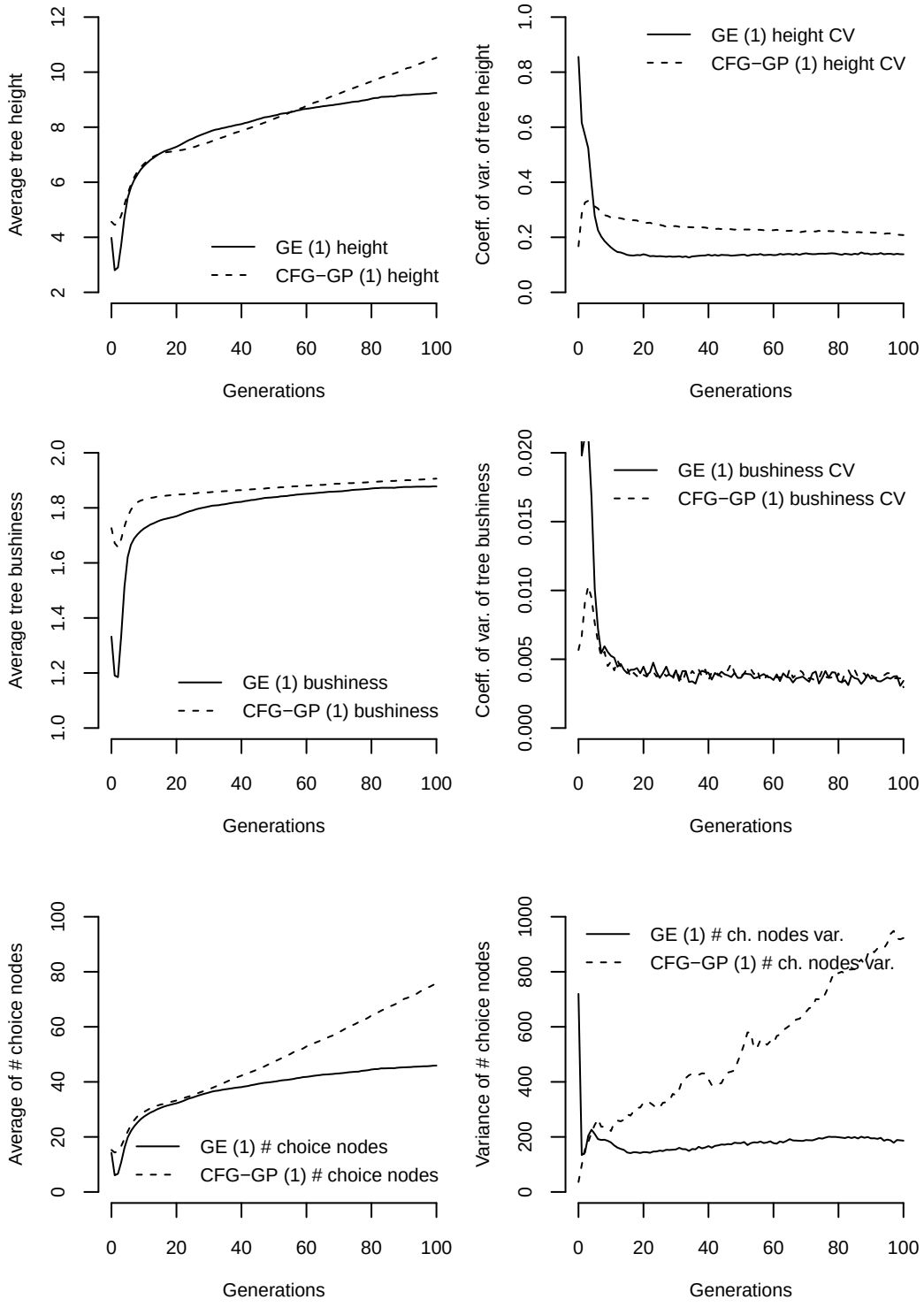


Figure 3.3: Tree characteristics of GE and CFG-GP in symbolic regression of the polynomial $x^4 + x^3 + x^2 + x$, setups 1. The plots show averages, taken over 1000 runs, of the following population statistics: averages and coefficients of variance of tree height and tree bushiness, average and variance of the number of choice nodes.

3.4.3 Experimental setups 2

We have used grammatical evolution with fixed chromosome length in the previous setup. GE was, however, originally conceived to be used with variable-length chromosomes (O'Neill and Ryan, 2003). We have also remarked that the initial population created by the random codon string initialisation has an extremely varied tree characteristics, which persist over several generations. This time, we will use variable chromosome length in conjunction with two different initialisation methods to find out how they affect the success rate and tree shape diversity. The first method is random codon string initialisation as in setups 1 but now with variable string length, and the second method is the "grow" method analogous to the one normally used in CFG-GP only with random genetic degeneracy added to each codon as usual in GE. Thus, using the "grow" method with maximum height 5, GE will now start with statistically equivalent phenotypes (trees and strings) to those CFG-GP started with in the first setup.

As before, we will use the same values of the remaining parameters for both methods when possible. As we have already evaluated CFG-GP with maximum initialisation height 5, we will now use two different maximum heights, 2 and 7, to assess the sensitivity of CFG-GP to characteristics of the initial population. (Derivation trees of height 2 are the shortest possible and can represent only the two strings `1.0` and `x`.) Settings used for both methods are listed in Table 3.2.

GE initialisation:	*GE (2a)*: Random codon string initialisation, length: 100–150. *GE (2b)*: "Grow" method (unique trees, random codon-level degeneracy), maximum height: 5.
GE operators:	Variable-length one-point crossover, probability: 0.9. Mutation as before.
CFG-GP initialisation:	*CFG-GP (2a)* and *(2b)*: "Grow" method (Whigham, 1995), maximum height: 2 and 7, respectively.
CFG-GP operators:	Crossover and mutation as before.

Table 3.2: Simple symbolic regression, parameters for setups 2, *CFG-GP (2a, 2b)*, and *GE (2a, 2b)*, where different from setups 1 (Table 3.1).

3.4.4 Results from setups 2

Both GE settings were an improvement over setups 1. The use of variable-length chromosomes alone in *GE (2a)* resulted in a modest increase to 807 successful runs, while the use of the CFG-GP-like initialisation in *GE (2b)* in a marked increase to 923 successful runs. Along with this improvement the tree-aware initialisation eliminated the extreme variation of tree characteristics at the start of the run. Another remarkable difference between the results of the two initialisation methods can be seen in the fitness statistics (Figure 3.5). The

use of the random string initialisation in conjunction with variable-length chromosomes resulted in frequent occurrence of highly unfit individuals throughout the run (the plot had to be cropped). While such individuals could have in principle occurred using any settings of both methods (they just represent relatively complex arithmetic expressions with output very far from the target function), the occurred only in GE with variable-length chromosomes and random-string initialisation (a combination used by O'Neill and Ryan, 2003, although they acknowledge the positive effect of tree-based initialisation).

CFG-GP with both initialisation parameters was still better than GE. With maximum initialisation height 2 it reached full success rate already in generation 58, and with maximum initialisation height 7 it succeeded in 992 runs out of 1000. The success rates for heights 2, 5, and 7 are significantly different, yet all three have led to successful solution quite consistently compared to GE.

We can make almost identical observations as in setups 1 about tree characteristics of the two methods: regardless of the initialisation method, GE results in bushier trees of more varied height and is able to expand the size of its search space over that of GE, at the same time CV of bushiness is similar for the two methods, and achieved tree heights can be too. This shows a consistent difference between GE and CFG-GP not caused by initialisation or use of variable-length vs. fixed-length chromosomes in GE, that is a difference resulting from the nature of operators of the two methods. The CFG-GP operators have searched the solution space of this simple problem more effectively than those of GE; additionally, CFG-GP has been relatively insensitive to the characteristics of the initial population.

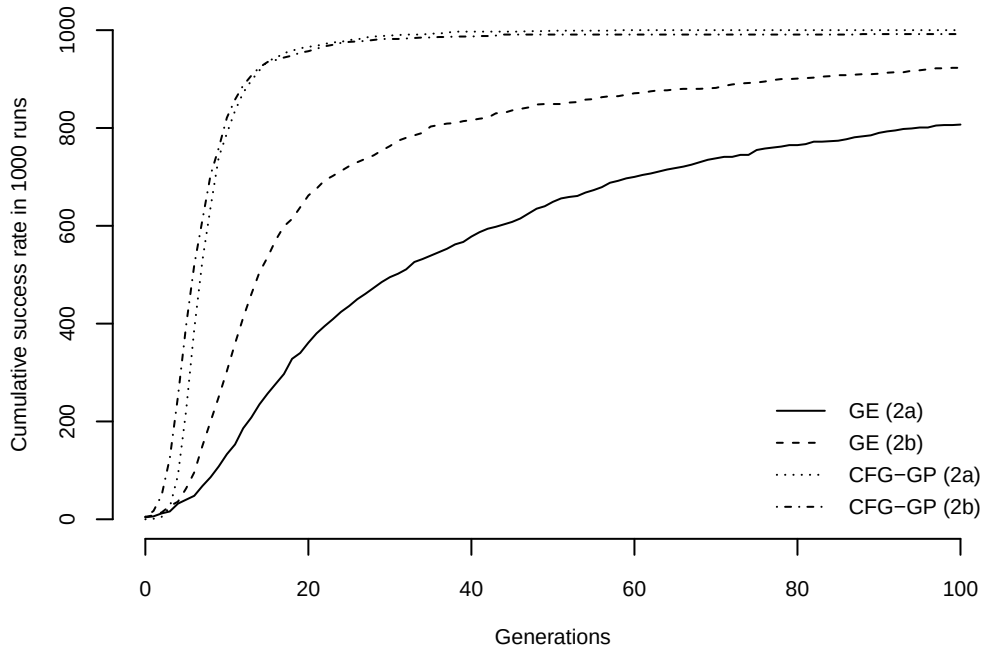


Figure 3.4: Cumulative frequency of success of GE and CFG-GP in symbolic regression of the polynomial $x^4 + x^3 + x^2 + x$, setups 2.

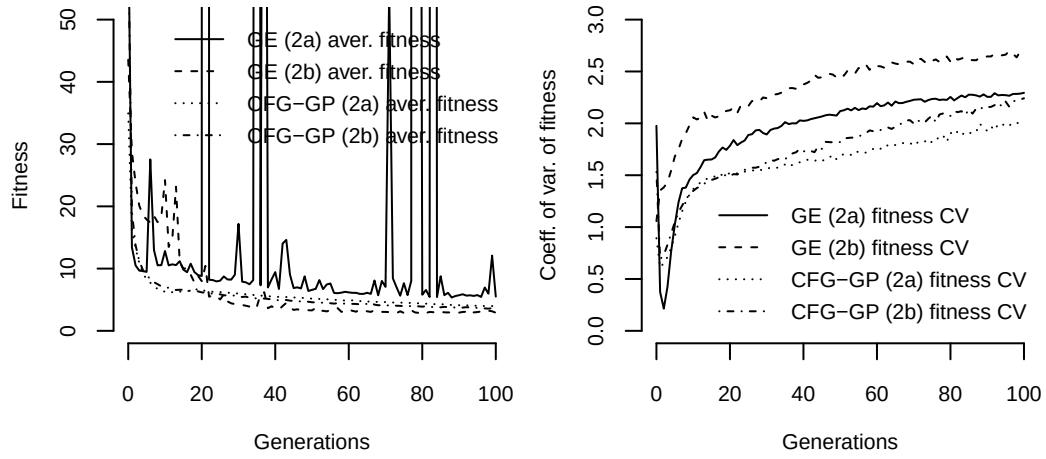


Figure 3.5: Fitness in GE and CFG-GP in symbolic regression of the polynomial $x^4 + x^3 + x^2 + x$, setups 2. The plots show averages, taken over 1000 runs, of the following population statistics: averages, and coefficients of variance of fitness. The fitness plot was cropped (inclusion of outliers would make it difficult to discern the important differences)

3.4.5 Conclusion

The presented problem of simple symbolic regression should indeed be simple to solve even with the relatively small population and number of generations: the grammar is relatively constrained and thus embeds significant amount of knowledge about the solution. The experiments have shown that CFG-GP can search the small space very effectively, achieving nearly full success rate regardless of selection method and initial population characteristics, whereas the success rate of grammatical evolution has been conspicuously low in comparison with CFG-GP despite improvements brought by a more appropriate selection method and the use of variable-length chromosomes.

Further analysis of characteristics of derivation trees during the run have shown that GE explores the same search space less effectively: its populations are less diverse than those evolved by CFG-GP. This is the result of different representation and operators employed by the two methods, and it is likely the cause of the disappointing performance of GE in this simple benchmark problem.

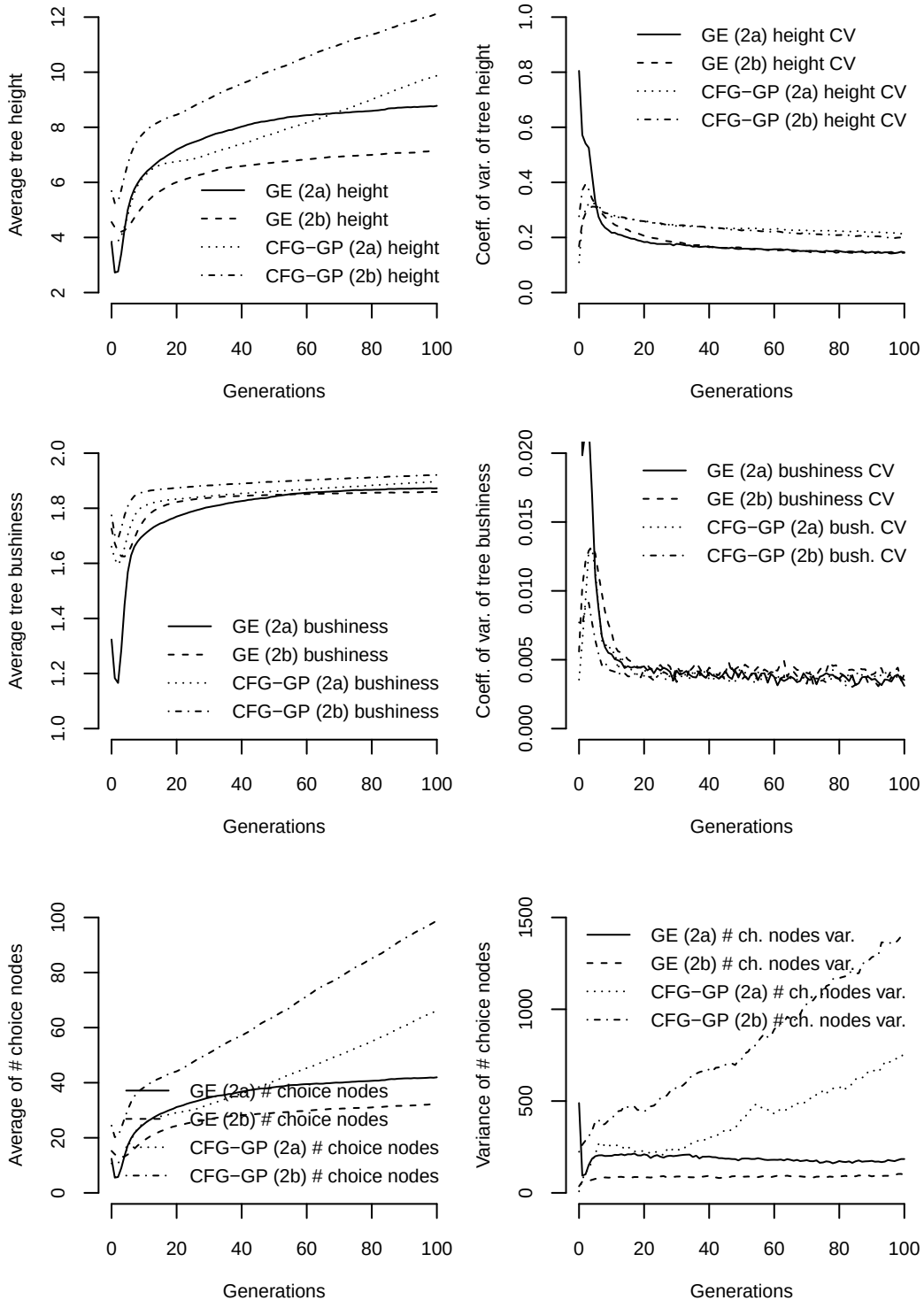


Figure 3.6: Tree characteristics of GE and CFG-GP in symbolic regression of the polynomial $x^4 + x^3 + x^2 + x$, setups 2. The plots show averages, taken over 1000 runs, of the following population statistics: averages and coefficients of variance of tree height and tree bushiness, average and variance of the number of choice nodes.

3.5 Santa Fe ant trail

In this experiment with the Santa Fe ant trail (described in Section 2.2) we will again start with a setup based upon an example GE application as implemented and configured in GEVA. Although the success of GE will initially be poor, it should be noted that this is not due to the implementation (as I have demonstrated in my bachelor thesis, Nohejl, 2009, GEVA achieves similarly poor results: in fact, AGE achieved exactly the same level of success when using the same initialisation method). All setups will use a relatively small population (100) and number of generations (101), so it is very likely that no parameters for GE and CFG-GP would result in full success. This way we will be able to compare population characteristics more clearly. In the symbolic regression experiment, CFG-GP converged so fast, that only a relatively small portion of the statistics concerned what happens until the solution is found.

3.5.1 Experimental setups 1

We will use the standard Santa Fe ant trail map in conjunction with a 600-tick time limit for each individual and the grammar in Listing 3.2 to replicate the configuration used in GEVA. Our goal is to compare different methods, not different grammars, but it is worth noting that the specification of grammar in this application offers room for subtle but important choices. The terminals simply represent the ant's actions and senses but we can decide how to combine them: for instance the grammar that we use does not allow nested conditional branching. We can see this as a bit of knowledge about the problem that has been embedded in the grammar: we (correctly) assume that nested conditional branching is not necessary for the solution. Additionally, the way the `<line>` and `<op>` nonterminals are chained in `<code>` and `<opcode>` productions will result in relatively high and sparse trees (compare to Listing 3.1 for symbolic regression, which can produce binary branching recursively). Both these features of the grammar result in a smaller search space and a faster search.

```
<prog>      ::= <code>
<code>      ::= <line> | <code> <line>
<line>      ::= <condition> | <op>
<condition> ::= if foodAhead(h)==1 then <opcode> else <opcode> end
<op>        ::= left(h) | right(h) | move(h)
<opcode>    ::= <op> | <opcode> <op>
```

Listing 3.2: A Lua equivalent to the Groovy grammar used for Santa Fe ant trail application included with GEVA.

In Section 3.4 we have suggested that different operators in CFG-GP and GE result in different spaces of tree shapes to be searched, but different initialisation techniques may have also played a role. This time, we will use the same two initialisation techniques with both methods. For *GE (1a)* and *CFG-GP (1a)*, we

will use the "sensible" initialisation as proposed by O'Neill and Ryan (2003) for GE. The "sensible" initialisation, being a variation on the ramped half-and-half initialisation (Koza, 1992), is a tree-based method that can be easily transposed from GE to CFG-GP. It is also used in the Santa Fe application in GEVA, although not implemented correctly (Nohejl, 2009, sec. 8.3.2). For *GE (1b)* and *CFG-GP (1b)* we will again use the same initialisation for both methods: "grow" initialisation as used by Whigham (1995) for CFG-GP. We will use the same maximum height value as for "sensible" initialisation, but the resulting average heights and distributions of tree shapes will be different.

Other parameters are listed in Table 3.3. Again, we use the same parameters whenever possible. In the case of mutation rate, which is measured differently for each method, we use a 0.01 per-codon rate for GE as in the setup used by GEVA and a 0.1 per-individual rate for CFG-GP as before (see Section 3.4.1 for details about the choice). We will try to assess the impact of mutation rates in the discussion of results.

Objective:	Find a program for navigating the ant so that it finds all 89 pieces of food on the Santa Fe trail within 600 time units.
Terminal operands:	None.
Terminal operators:	Left, Right, Move, Food-Ahead.
Grammar:	See Listing 3.2.
Fitness cases:	One fitness case.
Raw fitness:	Number of pieces of food left on the grid after 600 time units of running the ant's program in a loop.
Scaled fitness:	Same as raw fitness.
Algorithm:	Simple with elite size: 10, generations: 101, population: 100.
Selection:	Tournament, size: 3.
GE initialisation:	*GE (1a)*: Ramped ("sensible"), maximum height: 6. *GE (1b)*: "Grow" method (Whigham, 1995), maximum height: 6.
CFG-GP initialisation:	*CFG-GP (1a)*: Ramped ("sensible"), maximum height: 6. *CFG-GP (1b)*: "Grow" method (Whigham, 1995), maximum height: 6.
GE operators:	Variable-length one-point crossover, probability: 0.9. Codon-level mutation, probability: 0.01.
CFG-GP operators:	Crossover (Whigham, 1995), probability: 0.9. Mutation (Whigham, 1995), probability: 0.1.
Common parameters:	Maximum tree height: 20.
GE parameters:	Maximum wraps: 3. Codon size: 8.
Success predicate:	Raw fitness equals 0 (all food eaten).

Table 3.3: Santa Fe ant trail, parameters for setups 1, *GE (1a, 1b)*, *CFG-GP (1a, 1b)*.

3.5.2 Results from setups 1

Both methods were relatively unsuccessful. CFG-GP with either initialisation technique achieved a substantially higher success rate (191 and 206 out of 1000 runs were successful) than GE (9 and 24 runs out of 1000), see Figure 3.7. The differences caused by initialisation are small relatively to differences between GE and CFG-GP. The same pattern can be observed in tree characteristics except for bushiness where all differences are too small to draw a conclusion. Regardless of the initialisation method, the GE setups and likewise the CFG-GP setups converge to similar characteristics.

As in the previous setups for symbolic regression, in the long run CFG-GP with either initialisation method results in trees that are higher and more varied in height and number of choice nodes. The difference in success of the two methods is, however, much larger than before. Could this be attributed to a disparity in mutation rates? The coefficient of variance of fitness in the symbolic regression setups was either similar for both methods or higher for GE, now it is lower for GE (see Figure 3.8), which would suggest a lower effective mutation rate given no disparity in other parameters.

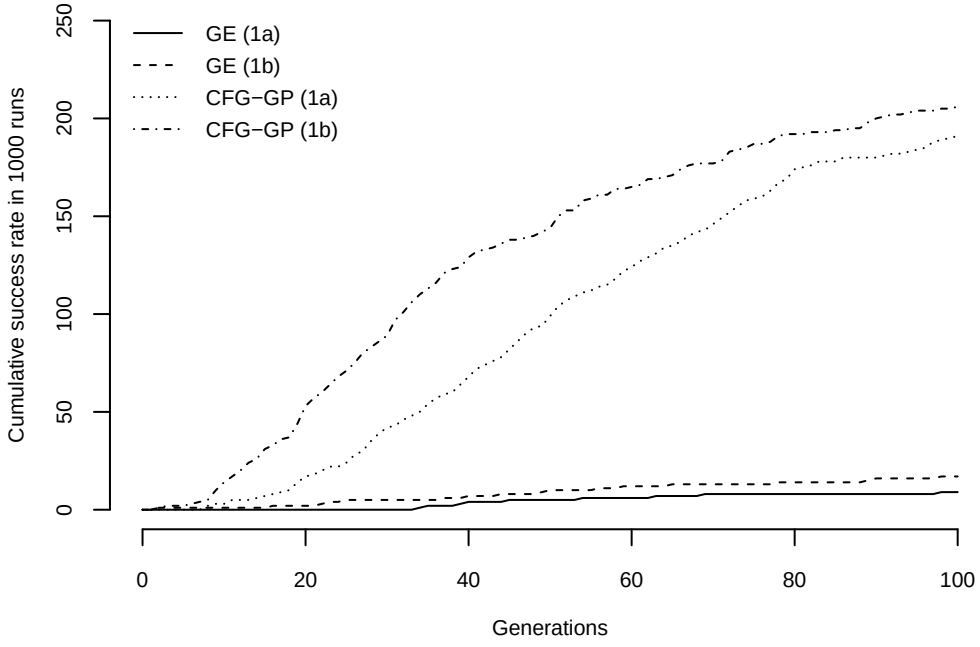


Figure 3.7: Cumulative frequency of success of GE and CFG-GP in Santa Fe ant trail, setups 1. (Scale from 0 to 250 out of 1000.)

We have observed the same kind of differences in success rates and tree characteristics between GE and CFG-GP as before but now independently of initialisation. Comparison of fitness statistics suggests, however, that part of the difference in success may be attributed to different effective mutation rates. Additionally, the less sophisticated "grow" method was more successful with both GE and CFG-GP. We will investigate the effect of initialisation using the second set of experimental setups.

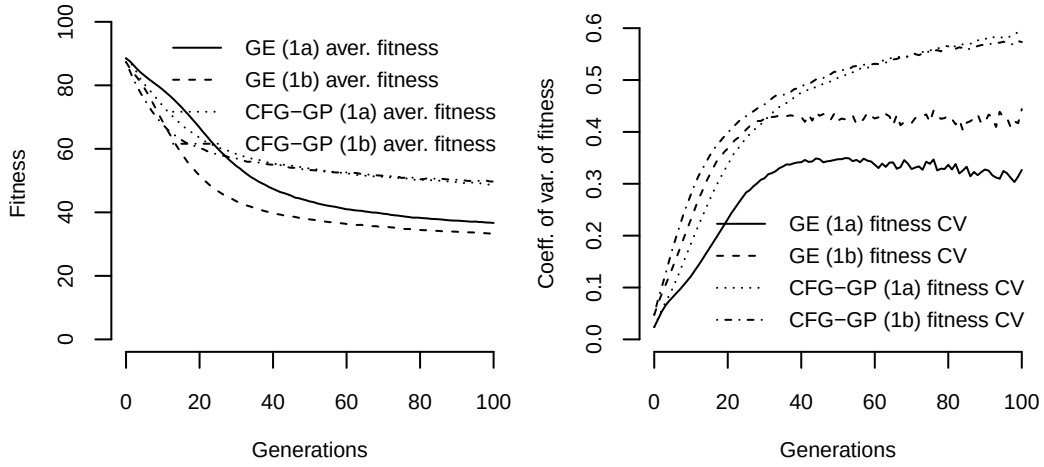


Figure 3.8: Fitness in GE and CFG-GP in Santa Fe ant trail, setups 1. The plots show averages, taken over 1000 runs, of the following population statistics: averages, and coefficients of variance of fitness.

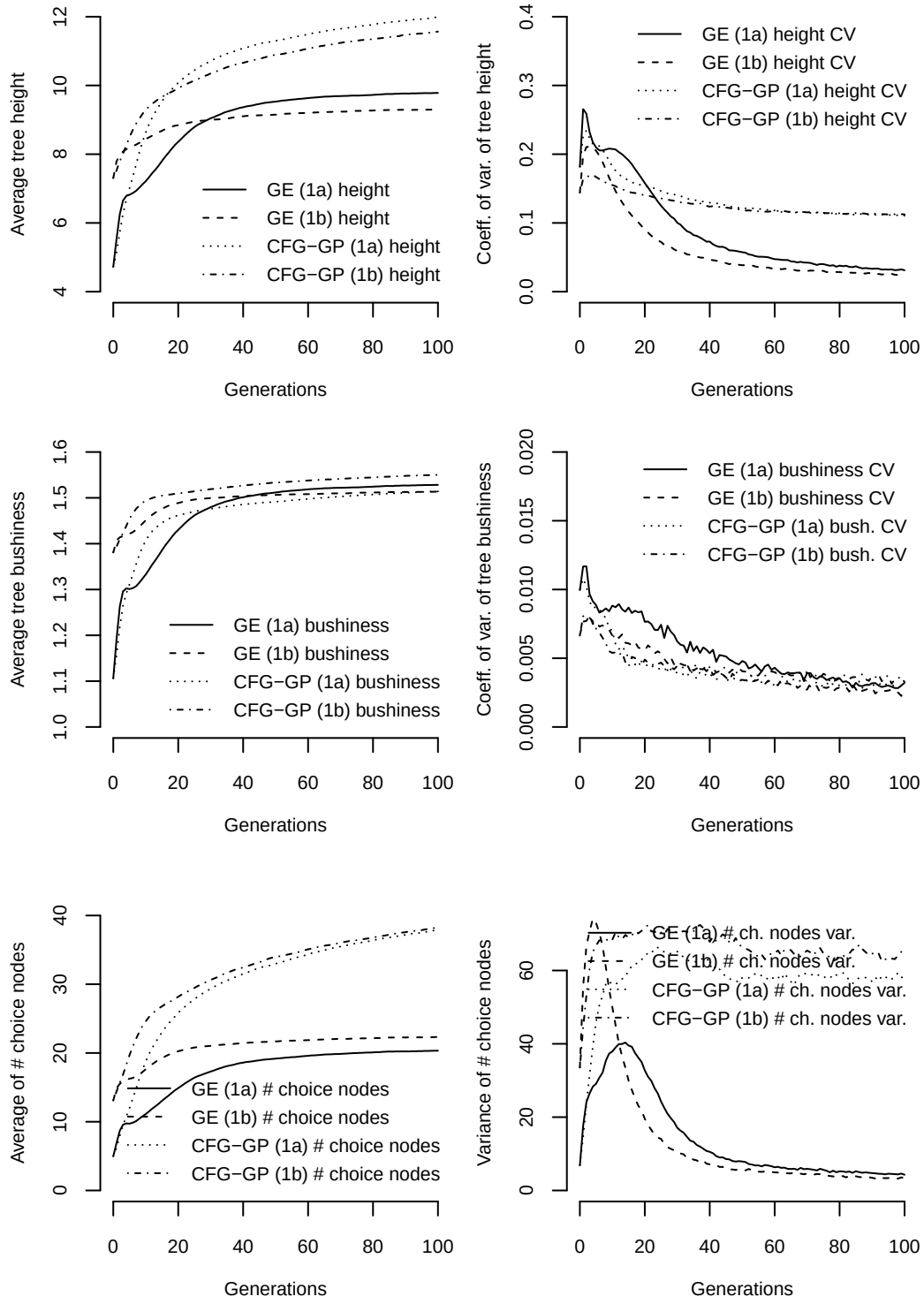


Figure 3.9: Tree characteristics of GE and CFG-GP in Santa Fe ant trail, setups 1. The plots show averages, taken over 1000 runs, of the following population statistics: averages and coefficients of variance of tree height and tree bushiness, average and variance of the number of choice nodes.

3.5.3 Experimental setups 2

In the second round we will use GE with bit-level, instead of codon-level, mutation rate 0.01 with the aim to increase the effective mutation rate. We expect this to cause GE to have comparable (or higher) CV of fitness than CFG-GP. We will also use a different initialisation method then in setups 1: the ramped half-and-half method modelled exactly after that introduced by Koza, 1992, which differs from the "sensible" initialisation (O'Neill and Ryan, 2003) most importantly by ensuring generation of unique trees at the expense of raising their height above the designated maximum (in the same way as the "grow" method used by Whigham, 1995). We suppose that this caused the relative success of the simpler "grow" method in setups 1. If this is correct, the ramped half-and-half initialisation should fare at least as well as the "grow" initialisation.

The differences from the previous setups are listed in Table 3.4.

GE initialisation:	Ramped half-and half (unique trees), maximum height: 6.
GE operators:	Crossover as before. Bit-level mutation, probability: 0.01.
CFG-GP initialisation:	Ramped half-and half (unique trees), maximum height: 6.
CFG-GP operators:	Crossover and mutation as before.

Table 3.4: Santa Fe ant trail, parameters for setups 2, *CFG-GP (2)*, and *GE (2)*, where different from setups 1 (Table 3.3).

3.5.4 Results from setups 2

The final success rates of CFG-GP and GE are 710 and 251 (see Figure 3.10), both several times higher than in the previous setups. The ramped half-and half initialisation has created trees of height only slightly lower than the "grow" initialisation but higher than the "sensible" initialisation. This shows that generation of unique trees, and the automatic increase of height it causes if needed, are vital parts of the ramped half-and-half initialisation. Without uniqueness it has perform worse than a simple "grow" method with uniqueness, while with uniqueness it performs substantially better.

The gap in success between CFG-GP and GE is also smaller than in the previous setups, and as we can see in Figure 3.11, the higher effective mutation rate has raised the CV of fitness for GE slightly above that of CFG-GP throughout the run.

Thus, we have evaluated GE and CFG-GP with similar effective mutation rates and the same initialisation with best results so far for both methods in the Santa Fe ant trail. The statistics of tree shapes of both methods are still related to each other in the same way as before and GE is still less successful.

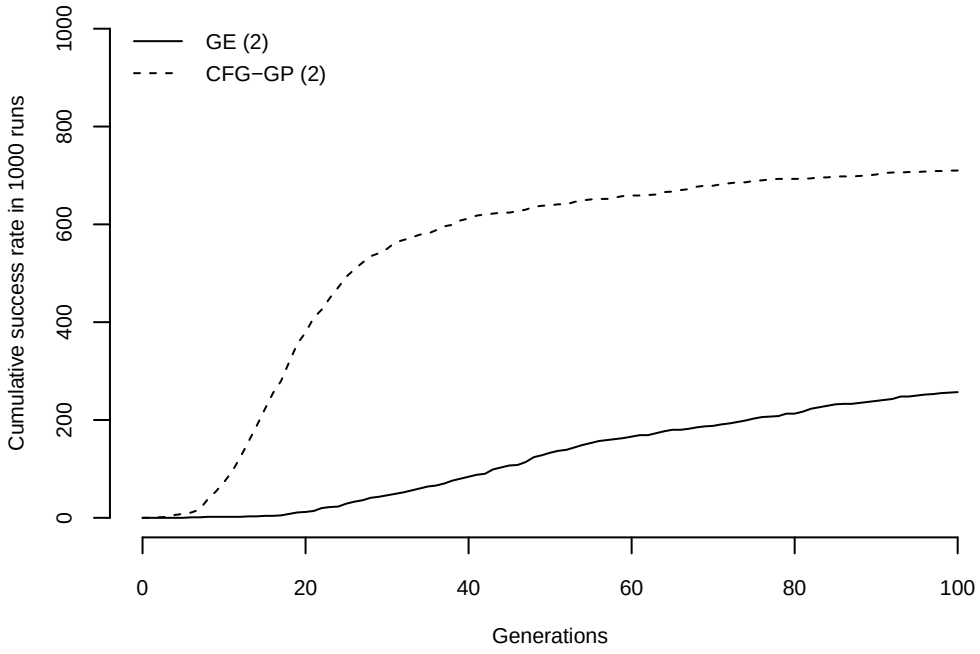


Figure 3.10: Cumulative frequency of success of GE and CFG-GP in Santa Fe ant trail, setups 2.

3.5.5 Conclusion

The Santa Fe ant trail problem differed from the simple symbolic regression. Foremost, it was more difficult to solve given the relatively small population size and number of generations. Also, the grammar we used was more restrictive in the sense that it excluded some constructs that traditional genetic programming would allow in this problem (nested if-then-else conditions) and that it did not allow recursive binary branching. Without recursive binary branching, trees are less bushy, and consequently there are not as many unique trees of a given height.

We have used the same parameters based on the setup used in GEVA for both GE and CFG-GP, including the same initialisation method ("sensible initialisation" proposed for GE by O'Neill and Ryan, 2003). We have also evaluated both methods with the "grow" initialisation (designed for use with CFG-GP by Whigham, 1995), and the ramped half-and-half initialisation (originally designed for tree-based GP by Koza, 1992). We have adjusted mutation rate for GE to reach similar effect in both methods (the mutation operators are qualitatively different, so we could do this only based on the CV of fitness). In all cases the methods displayed the same differences in tree characteristics we have already observed in simple symbolic regression (Section 3.4), which prevailed in the long run over the effect of initialisation. At the same time, we could see that both methods benefit from the same changes in initialisation. Koza's ramped half-and-half initialisation had the best results of the three tree-based techniques, partially thanks to the generation of unique individuals which automatically raises tree height as necessary.

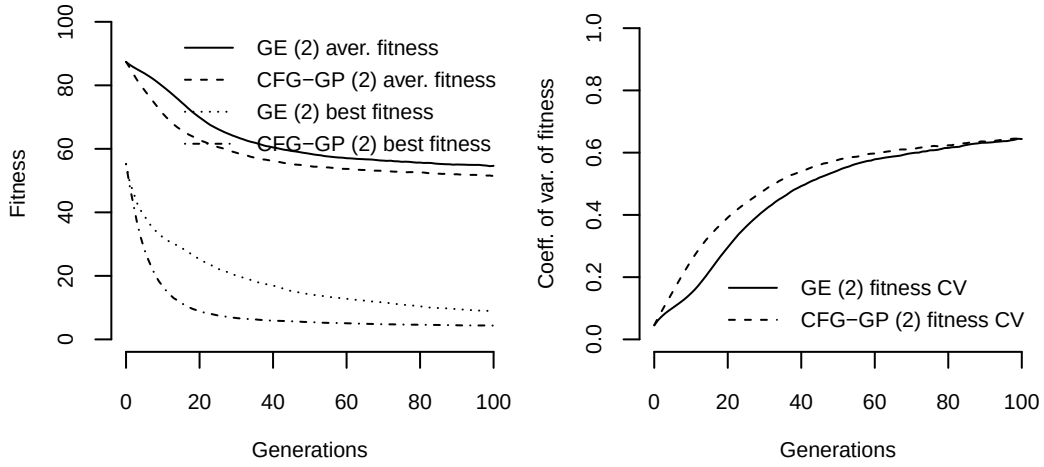


Figure 3.11: Fitness in GE and CFG-GP in Santa Fe ant trail, setups 2. The plots show averages, taken over 1000 runs, of the following population statistics: averages, minima (best values), and coefficients of variance of fitness.

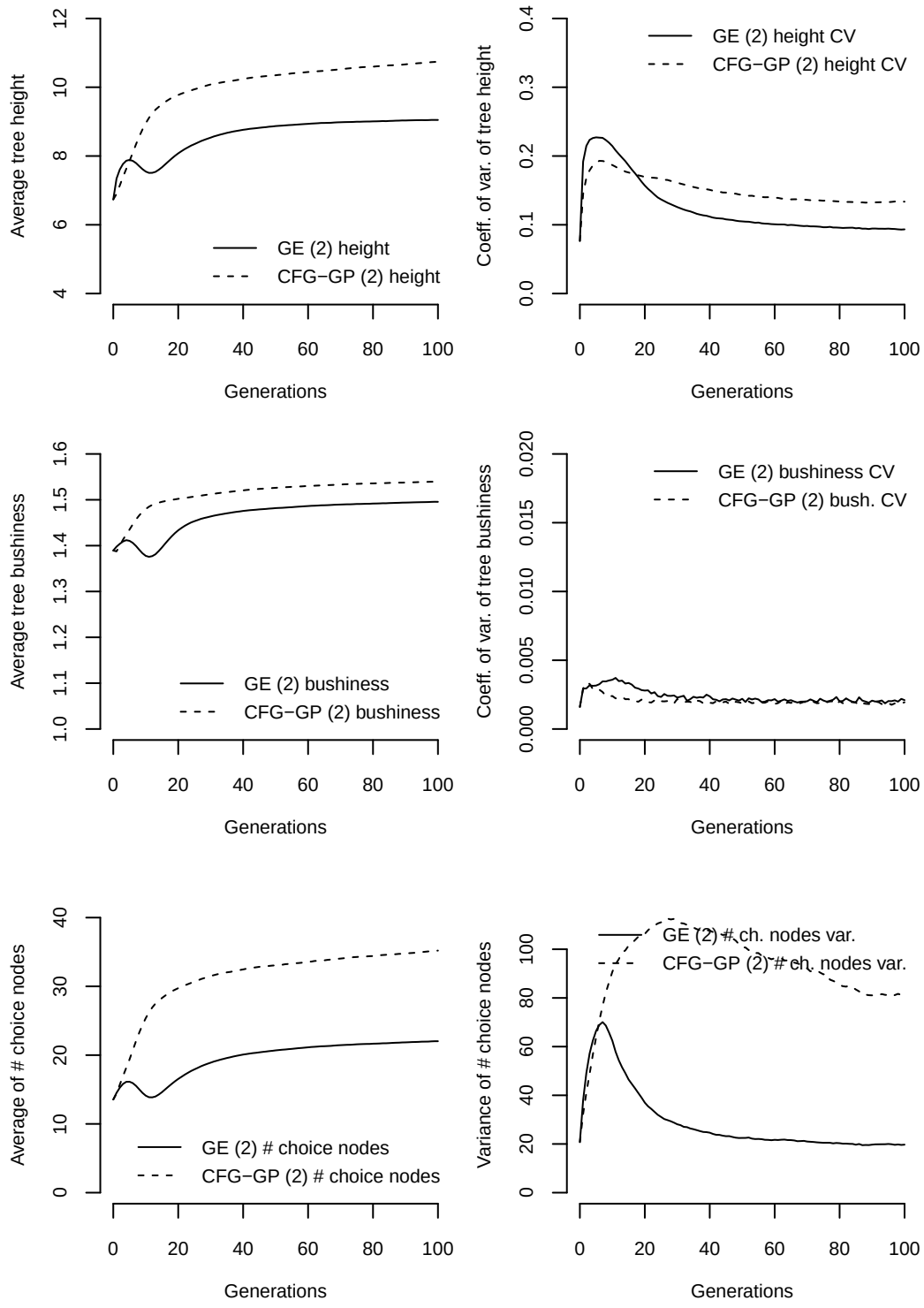


Figure 3.12: Tree characteristics of GE and CFG-GP in Santa Fe ant trail, setups 2. The plots show averages, taken over 1000 runs, of the following population statistics: averages and coefficients of variance of tree height and tree bushiness, average and variance of the number of choice nodes.

3.6 Dot product symbolic regression

In the previous experiments we have compared GE and CFG-GP as implemented in AGE, and we started with setups based on those used for GE in GEVA. In this section we will compare three methods: GE, CFG-GP, and LOGENPRO. The dot product problem and the initial setup we will use has been described by Wong and Leung (2000, sec. 6.1.2), who use it to demonstrate that LOGENPRO can express type constraints, thus outperforming basic tree-based GP, and generate ephemeral constants using logic goals (see Section 2.3).

```
start                      -> s-expr(number).
s-expr([list,number,?n]) -> [ (mapcar(function ], op2, [ ) ],
                              s-expr([list,number,?n]),
                              s-expr([list,number,?n]), [ ) ].
s-expr([list,number,?n]) -> [ (mapcar(function ], op1, [ ) ],
                              s-expr([list,number,?n]), [ ) ].
s-expr([list,number,?n]) -> term([list,number,?n]).
s-expr(number)             -> term(number).
s-expr(number)             -> [ (apply(function ], op2, [ ) ],
                              s-expr([list,number,?n]),[ ) ].
s-expr(number)             -> [ ( ], op2, s-expr(number),
                              s-expr(number), [ ) ].
s-expr(number)             -> [ ( ], op1, s-expr(number), [ ) ].
op2                        -> [ + ].
op2                        -> [ - ].
op2                        -> [ * ].
op2                        -> [ % ].
op1                        -> [ protected-log ].
term([list,number,n])      -> [ X ].
term([list,number,n])      -> [ Y ].
term(number)               -> { random(-10, 10, ?a) }, [ ?a ].
```

Listing 3.3: Logic grammar (LOGENPRO) for dot product symbolic regression in Lisp (from Wong and Leung, 2000). Unlike in a DCG, -> is the production symbol, logic variables are denoted by leading question marks, and terminals (in square brackets) do not need additional quotes.

To this end, Wong and Leung (2000) use an elaborate logic grammar (Listing 3.3) in a formalism based on DCGs. The terminals that contain calls to (1) `mapcar` and (2) `apply` are segments of Lisp code that apply the scalar operators to vectors (1) by components and (2) to components of a single vector, respectively. This is not something inherent to logic grammars, it is a convenient way to express different types of unary and binary operators in Lisp, which LOGENPRO uses to evaluate individuals.

What is interesting about the grammar is how it expresses type constraints. Two types are used: scalars and vectors of a given number of components. The

grammar expresses the types with arguments to functors `term`, and `s-expr`: `number` for scalars and `[list,number,?n]` or `[list,number,n]` for vectors. Seemingly, this takes advantage of the formalism of logic grammars and `n` could express the number of vector components, but upon further inspection, we see that all occurrences of the variable `n` (denoted `?n`) are unified with each other in every derivation, and in the end, via the two penultimate productions, also with the atom `n`. The compound term `[list,number,n]` does not have any inherent meaning, so all occurrences of `[list,number,?n]` and `[list,number,n]` could be replaced with an arbitrary atom (say `vector`) without any change to the semantics of the grammar. Therefore all rules without logic goals in the grammar could also be represented in a context-free grammar.

The last rule of the grammar contains a logic goal `random(-10, 10, ?a)`, which unifies the variable `a` with a random floating-point number between `-10` and `-10`. LOGENPRO evaluates logic goals only once: when generating the trees in initialisation or mutation, so this results in the same behaviour as the ephemeral random constants as used Koza (1992). This is the only case of actual use of the computational power of logic grammars demonstrated by Wong and Leung (2000). As the efficient evolution of ephemeral constants in GP is a nontrivial problem on its own (see Dempsey et al., 2009, ch. 5–6) and the solution to the dot product problem does not actually require constants, we will use the simple and obvious technique for CFG-GP and GE: there will be fixed nonterminals representing several arbitrarily chosen numbers from the target range. We will also evaluate LOGENPRO's performance with this simpler approach using a modified version of the original grammar (outlined inListing 3.4).

The grammar is obtained from Listing 3.3 by replacing the production

```
term(number)            -> { random(-10, 10, ?a) }, [ ?a ].
```

with the following 21 productions:

```
term(number)            -> [ -10 ].
term(number)            -> [ -9 ].
                           ⋮
term(number)            -> [ 9 ].
term(number)            -> [ 10 ].
```

Listing 3.4: Alternative logic grammar (LOGENPRO) for dot product symbolic regression in Lisp. Instead of generating random numbers via logic goals, it includes several arbitrary values as terminals.

This application is interesting chiefly as a demonstration of how grammars can efficiently describe type constraints. Once the constraints are enforced, it is a relatively easy problem: the tree height of the shortest solution is 5 and generating populations of 100 individuals using the "grow" method without uniqueness with maximum height 5 finds a solution in 241 cases out of 1000 (experiment performed in the same way as all experiments in the chapter).

3.6.1 Experimental setups

In the setup for LOGENPRO we will use the two presented grammars: the one used by Wong and Leung (2000), and the modified version for the sake of better comparison with CFG-GP and GE. Other parameters were chosen according to the description of the basic LOGENPRO algorithm and the original dot product experiment by Wong and Leung (2000, sec. 5.5, 6.1.2). The book, however, omits selection method (LOGENPRO uses either roulette-wheel or tournament selection), operator probabilities, and maximum tree height (for initialisation and during the run). I have chosen these parameters except the mutation rate according to the configuration files for the dot product problem provided to me along with the LOGENPRO source code by Dr Man Leung Wong. The mutation rate was set to 0.05 instead of 0 because:

– Wong and Leung (2000) present LOGENPRO as a method that uses two operators, crossover and mutation, evaluating a special case of the method would not be as valuable,

– LOGENPRO with mutation performed slightly better in this problem.

Additionally, I have redefined the `protected-log` function so that it returns 1 when undefined as specified by Wong and Leung (2000). (The source code supplied by Dr Wong defined the values for arguments less than 1 as 0.)

GE and CFG-GP were set up in the same way when possible. Both methods used the "grow" method without uniqueness, like LOGENPRO did, and the same grammar (Listing 3.5), which emulates the alternative grammar for LOGENPRO. For GE we have lowered the nominal mutation rate (similarly to the previous experiments). Table 3.5 contains the full parameters of the setups.

```
<start>    ::= return <num>
<num>      ::= <op2>v(<vec>)
           |   <op2>(<num>,<num>)
           |   <op1>(<num>)
           |   <num-term>
<vec>      ::= v<op2>(<vec>,<vec>)
           |   v<op1>(<vec>)
           |   <vec-term>
<op2>      ::= add | sub | mul | div
<op1>      ::= log
<vec-term> ::= x | y
<num-term> ::= -10|-9|-8|-7|-6|-5|-4|-3|-2|-1
           |   0|1|2|3|4|5|6|7|8|9|10
```

Listing 3.5: Context-free grammar in BNF for dot product symbolic regression in Lua. Vector-to-scalar operator names are suffixed by `v`. Vector operator names and prefixed by `v`. Scalar operators are named without suffix or prefix: `add`, `sub`, `mul`, `div`, `log`.

Objective:	Find a real-valued function of two three-component vector variables $\vec{x}$, $\vec{y}$ that computes the dot product $\vec{x} \cdot \vec{y}$.
Terminal operands:	$\vec{x}$, $\vec{y}$, ephemeral scalar constants between −10 and 10.
Terminal operators:	Addition, subtraction, multiplication, and division in three variants: scalar, vector (by components), vector-to-scalar (applied to components of a single vector); binary logarithm: scalar and vector (by components). Logarithm and division are protected (returning 1 when undefined).
Grammar:	See Listing 3.3 (original LOGENPRO grammar, results marked with *), Listing 3.4 (modified LOGENPRO grammar), Listing 3.5 (grammar for GE and CFG-GP).
Fitness cases:	10 vectors with components from $\{0, 1, 2, 3\}$ randomly generated for each run.
Raw fitness:	The sum of absolute errors taken over the 10 fitness cases.
Scaled fitness:	Same as raw fitness.
Algorithm:	Simple, generations: 100, population: 100.
Selection:	Tournament, size: 7.
LOGENPRO init.:	"Grow" method without uniqueness (Wong and Leung, 2000), maximum height: 6.
CFG-GP initialisation:	"Grow" method without uniqueness, maximum height: 6.
GE initialisation:	"Grow" method without uniqueness, maximum height: 6.
LOGENPRO operators:	Crossover (Wong and Leung, 2000), probability: 0.9. Mutation (Wong and Leung, 2000), probability: 0.05. Crossover and mutation are mutually exclusive.
CFG-GP operators:	Crossover (Whigham, 1995), probability: 0.9. Mutation (Whigham, 1995), probability: 0.05.
GE operators:	Fixed-length one-point crossover, probability: 0.9. Bit-level mutation, probability: 0.001.
Common parameters:	Maximum tree height: 9.
GE parameters:	Maximum wraps: 3. Codon size: 8.
Success predicate:	Raw fitness lower than 0.00001 (to allow for floating-point round-off error.)

Table 3.5: Dot product symbolic regression, parameters for LOGENPRO, CFG-GP, and GE. Note: Although the LOGENPRO implementation measures tree height by nodes instead of edges, the table follows the established convention (see page 5).

3.6.2 Results

The success rates of the four setups (Figure 3.13) are similar, often without a significant difference, only during the first thirteen generations. Thereafter, LOGENPRO with the original grammar is significantly less successful then any of the other three setups, LOGENPRO with the modified grammar is not significantly different from CFG-GP in generations 28 through 59, and CFG-GP is not significantly different from GE in generations 60 through 70, 75, 76, and then from 82 on. LOGENPRO with the original grammar, LOGENPRO with the modified grammar, CFG-GP, and GE have respectively succeeded in 53, 62, 68, and 68 of the 100 runs.

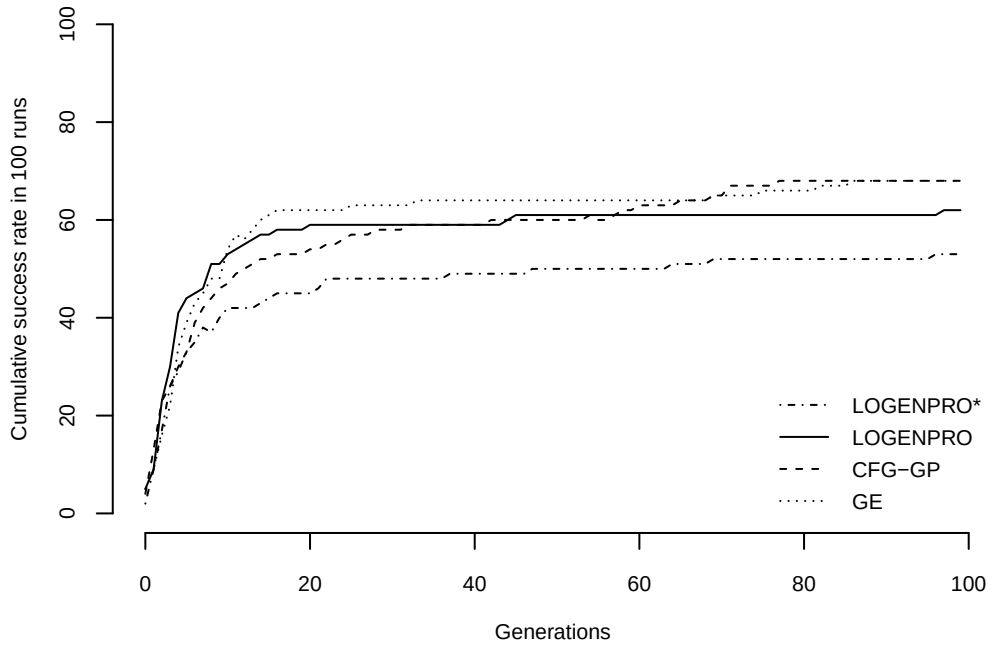


Figure 3.13: Cumulative frequency of success of LOGENPRO, CFG-GP and GE in symbolic regression of the dot product $\vec{x} \cdot \vec{y}$.

The most pronounced was the difference caused by the change of grammar in LOGENPRO, which can be attributed to the difference in search space size, given that ephemeral constants are of no use in this problem. As we could expect based on our discussion of the grammar used by LOGENPRO, there is no practical difference between the performance of LOGENPRO with a logic grammar and that of CFG-GP with a context-free grammar.

The 0.53 success rate (53 out of 100) obtained from LOGENPRO with the original grammar is lower than the 0.8 rate reported by Wong and Leung (2000, sec. 6.1.2, approximate reading of a plot). Regardless of the number of trials performed (not mentioned by Wong and Leung, 2000), such a difference would be statistically significant. I have not been able to replicate this success rate by varying operator probabilities, maximum tree heights or selection parameters, with either grammar. That said, even with the more modest success rate their comparison with tree-based GP would still hold (success rate of GP with the

same population size was reported below 0.1).

As LOGENPRO does not provide any statistics about the generated trees, we can compare these only for CFG-GP and GE, see Figure 3.14. In contrast with the previous experiments, in which CFG-GP performed better, GE has now reached exactly the same level of success, and in a large part of the run, there was no significant difference in success between the two. Despite these similar rates of success, we can still observe the differences in the tree characteristics: CFG-GP has soon reached both higher average values and CV of height and bushiness and maintained them throughout the run. As we have noted in the beginning of Section 3.6, correct solutions occur relatively densely among random trees of height 5, so the lower tree height close to 5 maintained by the operators of grammatical evolution (see Figure 3.14) are a very likely explanation for the improved success of GE.

3.6.3 Conclusion

We have shown that the use of logic grammars to describe types demonstrated by Wong and Leung (2000) does not require any of their distinctive features (unification and incorporation of logic goals): the same can be done using a context-free grammar in either CFG-GP or GE. Both methods have performed on a par with LOGENPRO (with a slightly higher success rate) using an equivalent context-free grammar and other parameters.

One could conceive of a type system that could be properly enforced using a logic grammar but not a context-free grammar. It is not obvious, however, that LOGENPRO would perform well if such a type system was necessary. It would depend on how efficiently the genetic operators would be able to fulfil their roles in the evolutionary algorithm under the more complex type constraints.

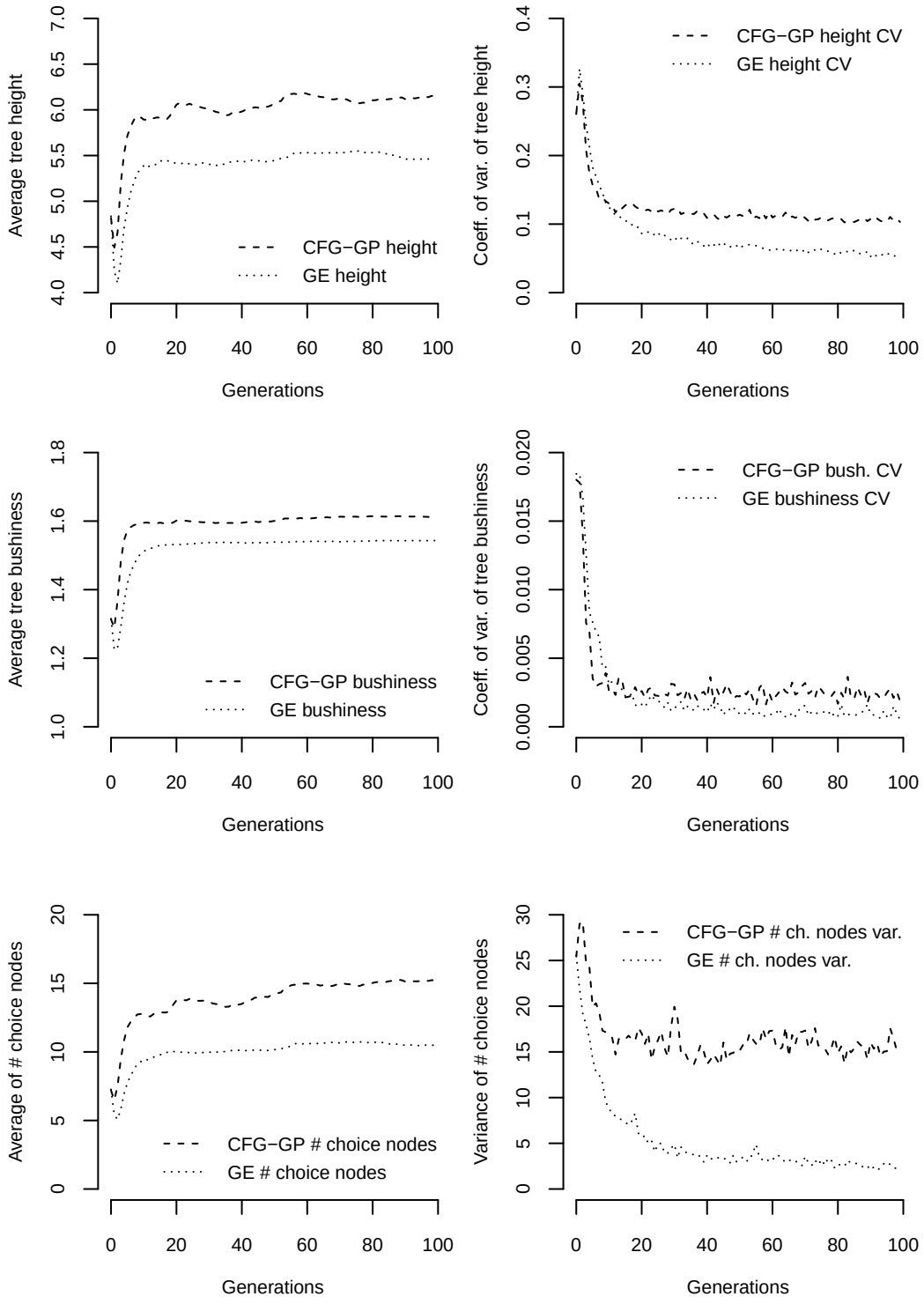


Figure 3.14: Tree characteristics of CFG-GP and GE in symbolic regression of the dot product. The plots show averages, taken over 100 runs, of the following population statistics: averages and coefficients of variance of tree height and tree bushiness, average and variance of the number of choice nodes.

3.7 Symbolic regression with ADFs

In this section, we will try to replicate another experiment presented by Wong and Leung (2000) using LOGENPRO, and compare the results with the other two grammar-based methods. It is again an instance of symbolic regression with multiple types but the target function is more complex. It has three vector variables $\vec{x}$, $\vec{y}$, $\vec{z}$, and can be expressed as $\vec{x} \cdot \vec{y} + \vec{y} \cdot \vec{z}$, where $\cdot$ is the dot product. To find an equivalent expression in a random search using the "grow" initialisation method with the grammar that we are going to use is an order of magnitude more difficult than it was to find the dot product. Wong and Leung (2000) designed the experiment to show that LOGENPRO can use ADFs similarly as traditional genetic programming (see Section 2.3).

To emulate ADFs, the grammar employed by Wong and Leung (2000) (Listing 3.6) uses two sets of nonterminals: those for the main program (`s-expr(···)`, `term(···)`) and those for the body of the ADF (`s-expr2(···)`, `term2(···)`). This way the terminals `X`, `Y`, `Z`, `ADF0`, and `arg0`, `arg1` can be restricted to occur only in the main subtree and the ADF subtree, respectively, and only subtrees of the same kind can be combined via crossover, which is equivalent to how ADFs are handled in tree-based GP. The grammar does not depend on any specific features of logic grammars (see explanation for a similar grammar in the previous experiment), so an equivalent CFG can be constructed (Listing 3.7) for use with GE and CFG-GP.

3.7.1 Experimental setups

As in Section 3.6, the setup of LOGENPRO replicates the one used by Wong and Leung (2000), and the missing parameters are chosen based on the configuration file obtained from Dr Wong. For the same reasons as before I also use the 0.05 mutation rate. Table 3.6 contains the full details of parameters for LOGENPRO, CFG-GP, and GE. The parameters are unchanged from the previous experiment without ADFs except for the number of generations lowered to 50 and a smaller number of terminals. Note that while the problem is obviously more difficult than the previous one, the search space has been restricted by removing several terminals from the grammar.

3.7.2 Results

All three methods have produced similar results in terms of success (see Figure 3.15): 51, 45, and 56 runs were successful out of 100 in the case of LOGENPRO, CFG-GP, and GE, respectively. There were significant differences in success between all of them in all generations of the second half of the run. In this experiment, grammatical evolution achieved the best performance of the three methods.

As in the previous experiment without ADFs, the success rate I have been able to achieve with LOGENPRO was significantly lower than that reported

```
start                        -> [ (progn (defun ADF0(arg0 arg1) ],
                                  s-expr2(number), [ ) ],
                                  s-expr(number), [ ) ].
s-expr([list,number,?n])     -> [ (mapcar(function ], op2, [ ) ],
                                  s-expr([list,number,?n]),
                                  s-expr([list,number,?n]), [ ) ].
s-expr([list,number,?n])     -> term([list,number,?n]).
s-expr(number)               -> [ (apply (function ], op2, [ ) ],
                                  s-expr2([list,number,?n]), [ ) ].
s-expr(number)               -> [ ( ], op2, s-expr(number),
                                  s-expr(number), [ ) ].
s-expr(number)               -> [ (ADF0 ],
                                  s-expr([list,number,?n]),
                                  s-expr([list,number,?n]), [ ) ].
term([list,number,n])        -> [ X ].
term([list,number,n])        -> [ Y ].
term([list,number,n])        -> [ Z ].
s-expr2([list,number,?n]) -> [ (mapcar(function ], op2, [ ) ],
                                  s-expr2([list,number,?n]),
                                  s-expr2([list,number,?n]), [ ) ].
s-expr2([list,number,?n]) -> term2([list,number,?n]).
s-expr2(number)              -> [ (apply(function ], op2, [ ) ],
                                  s-expr2([list,number,?n]), [ ) ].
s-expr2(number)              -> [ ( ], op2, s-expr2(number),
                                  s-expr2(number), [ ) ].
term2([list,number,n])       -> [ arg0 ].
term2([list,number,n])       -> [ arg1 ].
op2                          -> [ + ].
op2                          -> [ - ].
op2                          -> [ * ].
```

Listing 3.6: Logic grammar (LOGENPRO) for symbolic regression of the expression $\vec{x} \cdot \vec{y} + \vec{y} \cdot \vec{z}$ in Lisp (from Wong and Leung, 2000).

by Wong and Leung (2000, sec. 6.1.3). A rate higher than 0.65 can be read from their plot but even the lower one we have reached would be high enough to outperform GP without any constraints, for which they report a success rate lower than 0.1.

We can observe the same relationship between tree characteristics of GE and CFG-GP as in the previous experiment.

Wong and Leung (2000) do neither offer any details about the solutions found with their setup, nor do they compare the results with a LOGENPRO setup without ADFs. It is however rather conspicuous that the target expression $\vec{x} \cdot \vec{y} + \vec{y} \cdot \vec{z}$ can be rewritten more succinctly as $\vec{y} \cdot (\vec{x} + \vec{z})$, where the dot product appears only once. One would not expect a strong selection pressure

```
<start>     ::= function adf0(arg0,arg1) return <num'> end
                return <num>
<num>       ::= <op2>v(<vec>)
            |   <op2>(<num>,<num>)
            |   adf0(<vec>,<vec>)
<vec>       ::= v<op2>(<vec>,<vec>)
            |   <vec-term>
<num'>      ::= <op2>v(<vec'>)
            |   <op2>(<num'>,<num'>)
<vec'>      ::= v<op2>(<vec'>,<vec'>)
            |   <vec'-term>
<vec-term>  ::= x    | y    | z
<vec'-term> ::= arg0 | arg1
<op2>       ::= add  | sub  | mul
```

Listing 3.7: Context-free grammar in BNF for symbolic regression of the expression $\vec{x} \cdot \vec{y} + \vec{y} \cdot \vec{z}$ in Lua. Vector-to-scalar operator names are suffixed by `v`. Vector operator names and prefixed by `v`. Scalar operators are named without suffix or prefix: `add`, `sub`, `mul`.

to evolve a building block that needs to be used a single time. Indeed, out of the 51 solutions found by the successful runs of LOGENPRO I have performed, 24 use ADFs. 21 out of the 24 ADFs perform just addition on the components of one of its arguments. Two ADFs out of the remaining three do compute the dot product, which is then used in an expression equal to $\vec{y} \cdot (\vec{x} + \vec{z})$, and the last remaining ADF computes a more complex expression $\vec{arg0} \cdot (\vec{arg1} - \vec{arg0})$, which is then used in a similar way. The experiments with GE and CFG-GP had the same outcome: ADFs are not used as intended in any of the 100 runs.

This suggest that the experiment devised by Wong and Leung (2000) is not suitable for demonstrating the ability to use ADFs. I have performed it again with CFG-GP and GE but with ADF-related structures removed from the grammar (see Figure 3.17). The removal of ADFs has improved performance significantly for CFG-GP (from 45 to 55 successful runs out of 100), and insignificantly for GE (from 56 to 57 successful runs). This reflects that GE operators are grammar-agnostic as a result of the genotype-phenotype distinction, and so crossover can occur between a function-defining subtree and a main subtree regardless of the grammar. Without ADFs, CFG-GP and GE achieved similar performance: there is a significant difference in generations 1 through 16 and then only in the last 7 generations.

3.7.3 Conclusion

We have discovered that the problem specified by Wong and Leung (2000) is not suitable for evaluating performance grammar-based GP methods with automatically defined functions as it can be solved more simply and efficiently

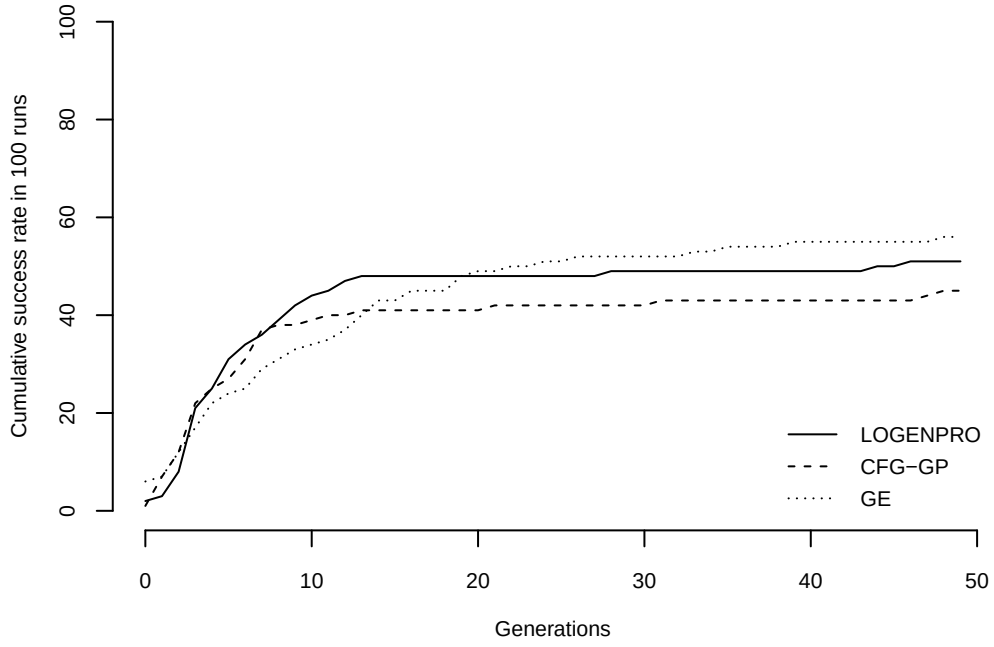


Figure 3.15: Cumulative frequency of success of LOGENPRO, CFG-GP, and GE in symbolic regression of the expression $\vec{x} \cdot \vec{y} + \vec{y} \cdot \vec{z}$.

without them. Nevertheless we have made three valuable observations. First, GE and CFG-GP have performed on a par with LOGENPRO. Secondly, as in the previous experiments, CFG-GP has in the long run produced more varied tree shapes, as characterised by bushiness and height. Thirdly, GE has suffered less from the unnecessary addition of ADF-related structures to the grammar because, in contrast with CFG-GP and LOGENPRO, the syntactical constraints do not constrain the operators, which act only on the genotype. In order to enforce similar behaviour in grammatical evolution, we would have to use separately mapped chromosomes for each function-defining branch and for each value-returning branch and allow crossover only between chromosomes of the same type.

Objective:	Find a real-valued function of three three-component vector variables $\vec{x}$, $\vec{y}$, $\vec{z}$ that yields $\vec{x} \cdot \vec{y} + \vec{y} \cdot \vec{z}$, where $\cdot$ is the dot product.
Terminal operands:	$\vec{x}$, $\vec{y}$, $\vec{z}$. Within ADF: $\vec{arg0}$, $\vec{arg1}$.
Terminal operators:	Addition, subtraction, and multiplication in three variants: scalar, vector (by components), vector-to-scalar (applied to components of a single vector). Out of ADF: the vector-to-scalar function *ADF0* of two arguments.
Grammar:	See Listing 3.6 (LOGENPRO), Listing 3.7 (GE and CFG-GP).
Fitness cases:	10 vectors with components from $\{0, 1, 2, 3\}$ randomly generated for each run.
Raw fitness:	The sum of absolute errors taken over the 10 fitness cases.
Scaled fitness:	Same as raw fitness.
Algorithm:	Simple, generations: 50, population: 100.
Selection:	Tournament, size: 7.
LOGENPRO init.:	"Grow" method without uniqueness (Wong and Leung, 2000), maximum height: 6.
CFG-GP initialisation:	"Grow" method without uniqueness, maximum height: 6.
GE initialisation:	"Grow" method without uniqueness, maximum height: 6.
LOGENPRO operators:	Crossover (Wong and Leung, 2000), probability: 0.9. Mutation (Wong and Leung, 2000), probability: 0.05. Crossover and mutation are mutually exclusive.
CFG-GP operators:	Crossover (Whigham, 1995), probability: 0.9. Mutation (Whigham, 1995), probability: 0.05.
GE operators:	Fixed-length one-point crossover, probability: 0.9. Bit-level mutation, probability: 0.001.
Common parameters:	Maximum tree height: 9.
GE parameters:	Maximum wraps: 3. Codon size: 8.
Success predicate:	Raw fitness lower than 0.00001 (to allow for floating-point round-off error.)

Table 3.6: Symbolic regression of the expression $\vec{x} \cdot \vec{y} + \vec{y} \cdot \vec{z}$, parameters for LOGENPRO, CFG-GP, and GE. Note: Although the LOGENPRO implementation software tree height by nodes instead of edges, the table follows the established convention (see page 5).

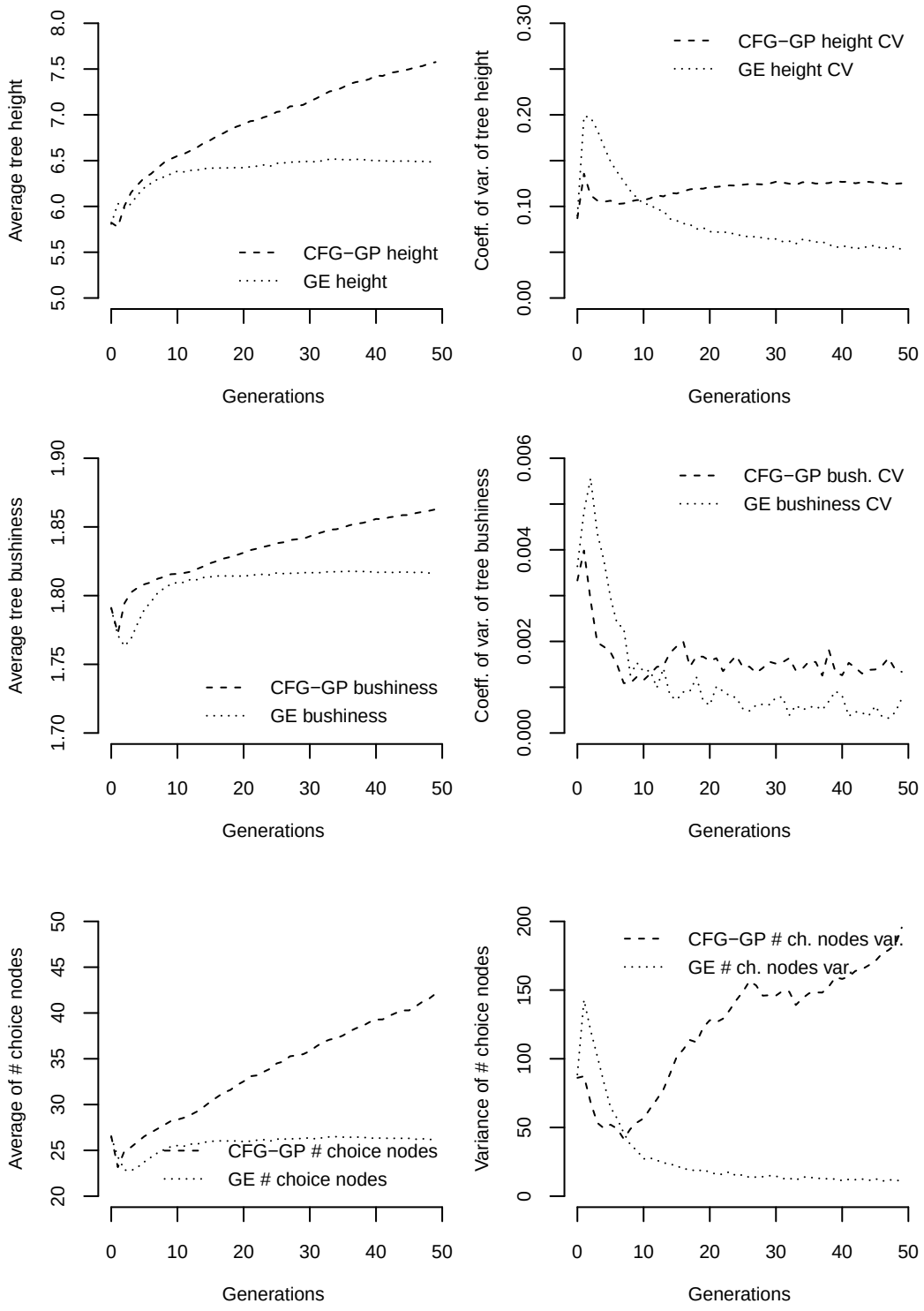


Figure 3.16: Tree characteristics of CFG-GP and GE in symbolic regression of the expression $\vec{x} \cdot \vec{y} + \vec{y} \cdot \vec{z}$. The plots show averages, taken over 100 runs, of the following population statistics: averages and coefficients of variance of tree height and tree bushiness, average and variance of the number of choice nodes.

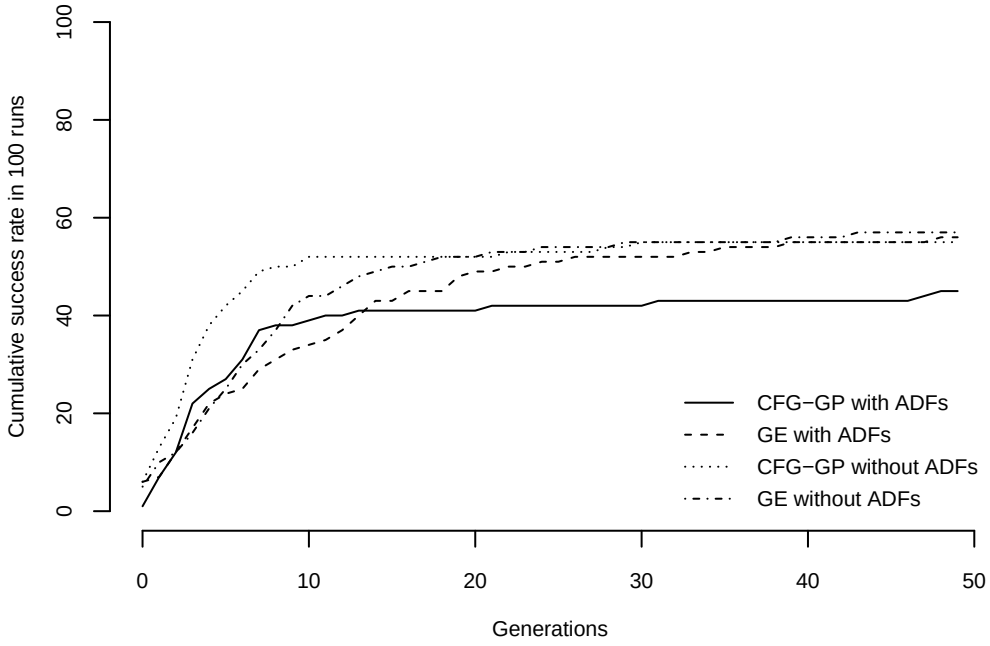


Figure 3.17: Cumulative frequency of success of CFG-GP and GE without ADFs in symbolic regression of the expression $\vec{x} \cdot \vec{y} + \vec{y} \cdot \vec{z}$.

3.8 Boolean parity functions with ADFs

In the previous experiment we have shown that the symbolic regression problem used by Wong and Leung (2000) to demonstrate the use of ADFs in LOGENPRO is not fit for the purpose. Additionally, the problem involves multiple types, so a grammar-based approach with a suitable grammar would always have an advantage over tree-based GP without type constraints, regardless of how it handled ADFs. Specifying ADFs using a context-free grammar in CFG-GP (or LOGENPRO) will have the same effect on operators as the constraints Koza (1992) has used with ADFs in tree-based GP. The operators used in grammatical evolution, however, operate on the genotype rather than on the grammatically constrained phenotype. In the previous experiment, we have observed that this is helpful, if the ADF-related structures are unnecessary, but we have yet to find out how well GE can perform when ADFs are actually needed.

To be able to compare performance of GE both with CFG-GP and tree-based GP we will use the problem of symbolic regression of the Boolean parity functions (see Section 2.4). We will essentially try to replicate the results achieved by Koza (1992) with GP in a series of experiments with these functions.

3.8.1 Experimental setups

We will perform a series of the following experiments:

– symbolic regression of the even-3-parity function without ADFs,
– symbolic regression of the even-4-parity function with and without ADFs,
– symbolic regression of the even-5-parity function with and without ADFs,
– symbolic regression of the even-6-parity function with ADFs.

There will be $n - 2$ function-defining branches for an n-ary parity function with ADFs. The two ADFs for the even-4-parity function will have 3 and 4 arguments. The three ADFs for the even-5-parity function will have 2, 3 and 4 arguments. The four ADFs for the even-6-parity function will have 2, 3, 4, and 5 arguments.

We will use the same parameters for CFG-GP as Koza (1992, sec. 20.1) did for GP. In order to make the setup as close as possible to the original setup for GP, we will use grammars that mimic the behaviour of tree-based GP by having one nonterminal for each GP node type, and a GP-like crossover operator, which operates on inner tree nodes with 90% probability. Koza did not use a mutation operator, but this variant of crossover effectively emulates a point mutation with 10% probability. Listing 3.8 and Listing 3.9 provide examples of the grammars we will use, the other grammars follow the same pattern. For GE, we will use its usual operators with conservatively chosen rates, the other parameters remain unchanged. The full setup for both methods is specified in Table 3.7.

Objective:	Find an expression for the even parity function of 3, 4, 5, and 6 arguments using ADFs as described in the text.
Terminal operands:	A subset of `d0`, `d1`, ..., `d5` (variables), `a0`, `a1`, ..., `d4` (ADF parameters) based on arity and whether ADFs are used.
Terminal operators:	Boolean connectives *and*, *or*, *nand*, *nor*; a subset of `adf0`, `adf1`, `adf2` based on arity and whether ADFs are used.
Grammar:	See Listing 3.8 and Listing 3.9 for examples.
Fitness cases:	The set of all possible arguments.
Raw fitness:	The number of fitness cases for which the value returned by the expression is different from the correct value of the even parity function.
Scaled fitness:	Koza's adjusted fitness, and greedy over-selection with the parameters used by Koza (1992).
Algorithm:	Simple, generations: 51, population: 4000.
Selection:	Roulette-wheel.
CFG-GP initialisation:	Ramped half-and half (unique trees), maximum height: 8/7 (with/without ADFs).
GE initialisation:	Ramped half-and half (unique trees), maximum height: 8/7 (with/without ADFs).
CFG-GP operators:	Only GP-like crossover (Koza, 1992), probability: 0.9.
GE operators:	Fixed-length one-point crossover, probability: 0.9 Bit-level mutation, probability: 0.002.
Common parameters:	Maximum tree height: 18/19 (with/without ADFs).
GE parameters:	Maximum wraps: 3. Codon size: 8.
Success predicate:	Raw fitness equal to 0.

Table 3.7: Symbolic regression of the Boolean parity function, CFG-GP, and GE. Note: the maximum initialisation tree height and the overall maximum tree height correspond to GP tree heights 6 and 17, respectively, used by Koza (1992).

3.8.2 Results

Figure 3.18 shows plots of cumulative frequency of success for all experiments. In Table 3.8 we compare the final success rates with those reported for tree-based GP by Koza (1992). Koza performed tens of runs in all experiments except for even-5-parity with ADFs (7 runs) and even-6-parity with ADFs (unreported). Both CFG-GP and GE perform surprisingly well without ADFs compared to the results reported by Koza (1992) for GP: for instance in the even-4-parity, CFG-GP has more than twice the success rate of GP, and even the even-5-parity without ADFs intimidates neither CFG-GP nor GE.

We are, however, more interested in the performance with ADFs. Context-free grammar genetic programming performs only slightly worse than tree-based genetic programming in the even-4-parity problem (96% vs. 99% success). For even-5-parity, Koza (1992) reports results from mere 7 runs, so I will avoid drawing any conclusion from the comparison: incidentally the first 7 runs I have performed with CFG-GP were also successful.) CFG-GP performs relatively well even in the even-6-parity problem (Koza does not report any

```
<start> ::= return <expr>
<expr>  ::= (<expr> and <expr>) | (<expr> or <expr>)
         |  not (<expr> and <expr>) | not (<expr> or <expr>)
         |  d0 | d1 | d2
```

Listing 3.8: Context-free grammar in BNF for symbolic regression of a ternary Boolean function in Lua. The connectives nand and nor are expressed using the three basic connectives not, or, and.

```
<start> ::= function adf0(a0,a1) return <expr0> end
            function adf1(a0,a1,a2) return <expr1> end
            return <expr>
<expr>  ::= (<expr> and <expr>) | (<expr> or <expr>)
         |  not (<expr> and <expr>) | not (<expr> or <expr>)
         |  adf0(<expr>,<expr>) | adf1(<expr>,<expr>,<expr>)
         |  d0 | d1 | d2 | d3
<expr0> ::= (<expr0> and <expr0>) | (<expr0> or <expr0>)
         |  not (<expr0> and <expr0>)
         |  not (<expr0> or <expr0>) | a0 | a1
<expr1> ::= (<expr1> and <expr1>) | (<expr1> or <expr1>)
         |  not (<expr1> and <expr1>)
         |  not (<expr1> or <expr1>) | a0 | a1 | a2
```

Listing 3.9: Context-free grammar in BNF for symbolic regression of a quaternary Boolean function using ADFs in Lua.

precise results). Overall, the results of CFG-GP with ADFs are similar to those of tree-based GP as far as we can judge from the results provided by Koza (1992).

Grammatical evolution performs substantially worse with ADFs than CFG-GP. While it reached a significantly lower success rate also without ADFs, the difference between CFG-GP and GE was not that marked. Clearly, ADF still improve the performance of GE in this problem, but not as effectively as in the case of CFG-GP and tree-based GP. This is likely a result of the different effect grammar-defined ADFs have on the operators of CFG-GP and GE, as we have already noted.

Figure 3.19 shows plots of tree characteristics in the even-5-parity with ADFs experiment. This time, GE produces trees of higher and more varied tree bushiness and also more varied height. This is not very surprising because we have set up CFG-GP to use the GP-like crossover operator and no mutation, which is normally responsible for generation of new subtrees. Nevertheless, CFG-GP creates much larger trees thanks to a steadily growing tree height, while the operators of GE do not allow higher and larger trees to evolve. (This also applies to tree characteristics produced by the two methods in the other experiments. The plots are omitted for brevity.)

Method	GP		CFG-GP		GE	
ADFs	✗	✓	✗	✓	✗	✓
even-3-parity	100 %	–	100 %	–	93 %	–
even-4-parity	45 %	99 %	93 %	96 %	25 %	77 %
even-5-parity	†0 %	‡7 of 7	7 %	72 %	8 %	37 %
even-6-parity	–	*> 0	–	44 %	–	22 %

†: no success in 20 runs. ‡: 100% success but only in 7 runs.
*: a solution can be found within 20 runs.

Table 3.8: Comparison of success rates of tree-based GP, CFG-GP, and GE in the even parity problems.

3.8.3 Conclusion

In the series of experiments with symbolic regression of the Boolean even parity functions using ADFs, CFG-GP has achieved success rates comparable with those reported by Koza (1992) for tree-based GP. The success rates of GE were substantially lower, but nevertheless ADFs improved its performance. We can conclude that the grammar-based ADFs we have used are also suitable for grammatical evolution, but if we know beforehand that ADFs will be needed for some problem, tree-based GP or CFG-GP will likely provide a superior performance. This reveals what might be a general problem with GE: if individuals consist of subtrees with terminals of mutually incompatible types, we can expect the crossover operator to perform poorly in GE.

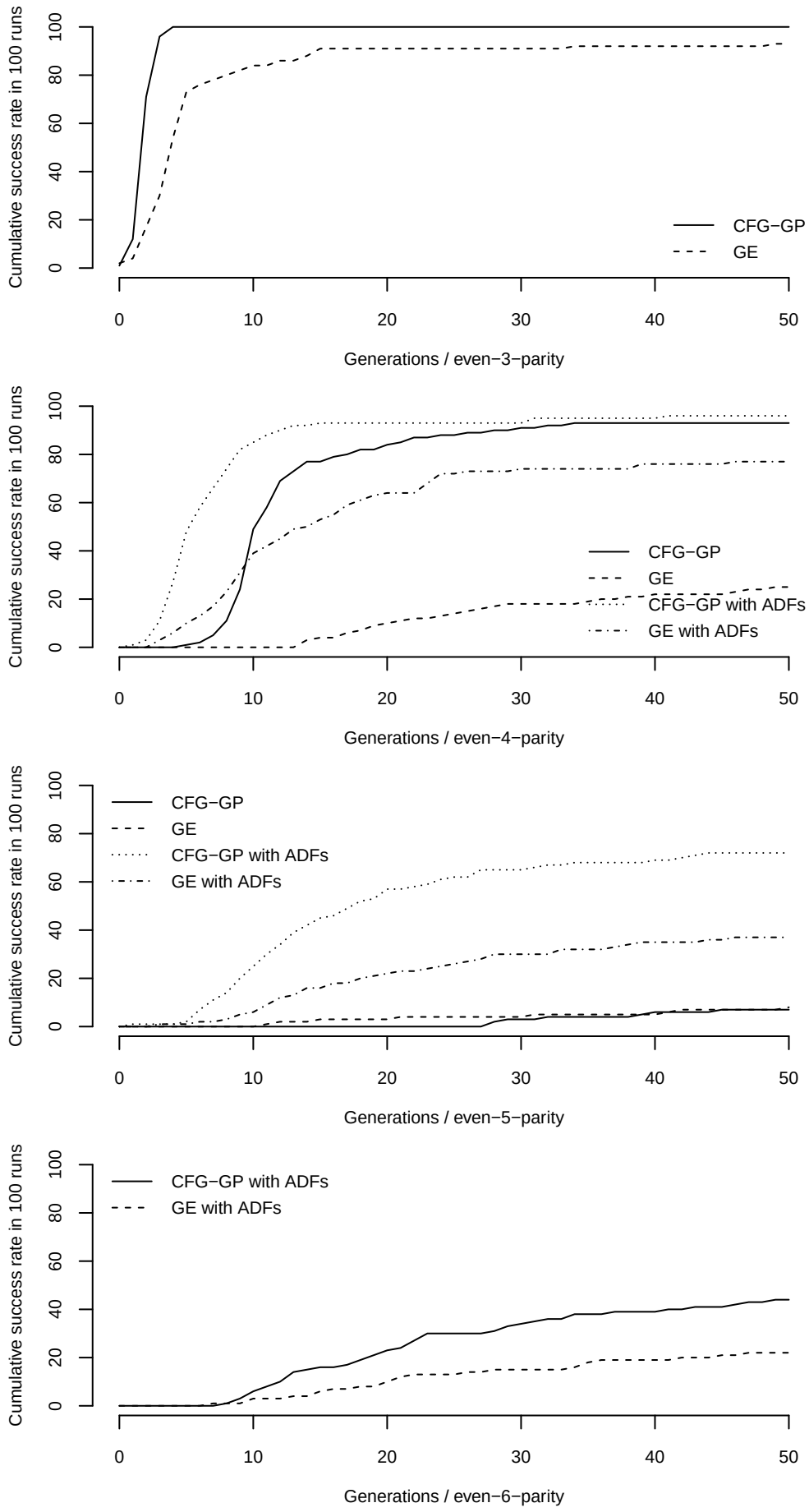


Figure 3.18: Cumulative frequency of success of CFG-GP and GE with and without ADFs in symbolic regression of the Boolean even parity functions.

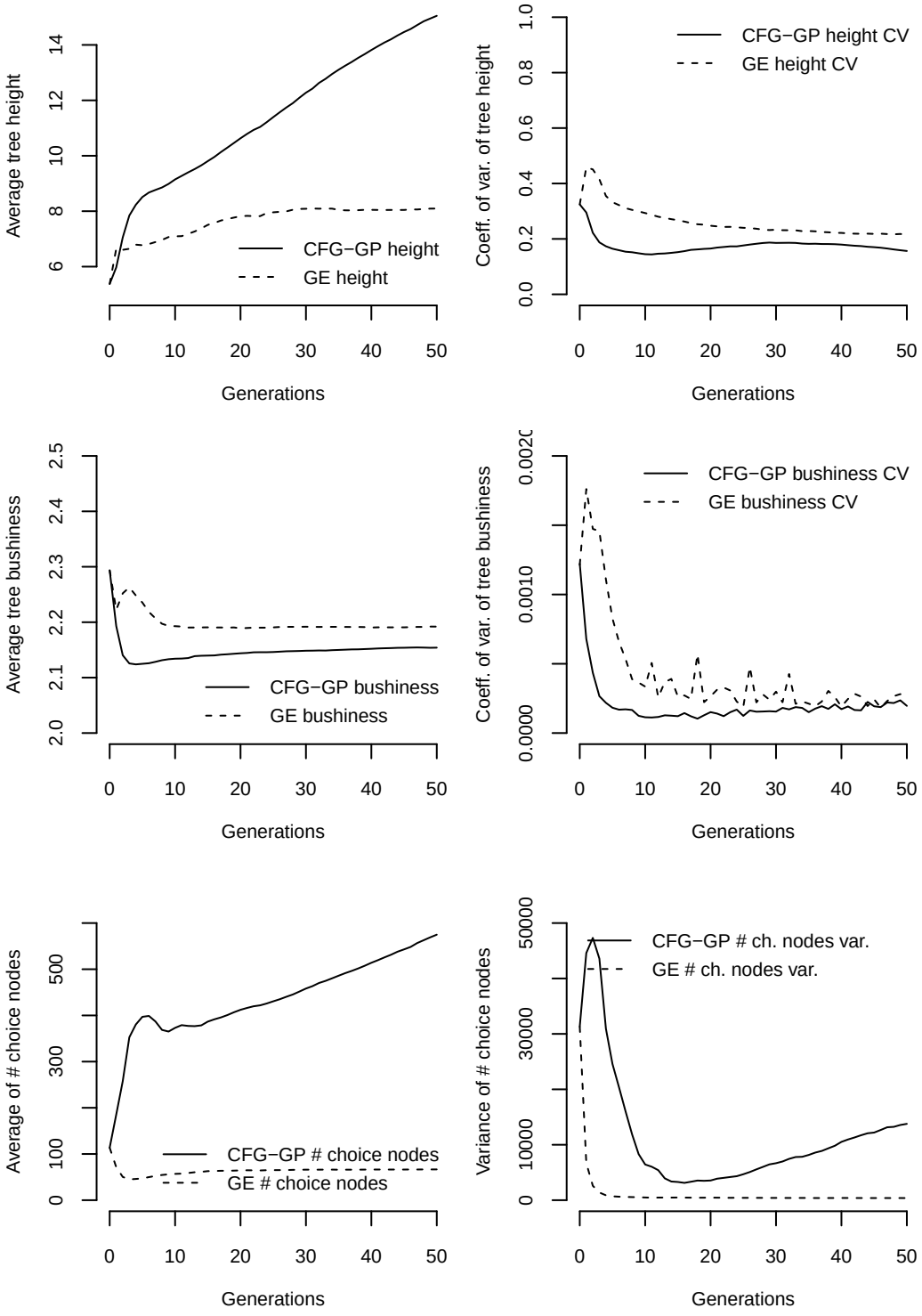


Figure 3.19: Tree characteristics of CFG-GP and GE with ADFs in symbolic regression of the Boolean even-5-parity function. The plots show averages, taken over 100 runs, of the following population statistics: averages and coefficients of variance of tree height and tree bushiness, average and variance of the number of choice nodes.

3.9 Exam timetabling hyper-heuristics

Exam timetabling is a real-world problem of scheduling exams for university courses under given constraints. The constraints may vary according to institution but the most usual hard constraint is that no student can attend two exams at the same time, and the most usual soft constrain is that exams scheduled very close to each other for some students should be avoided. Most literature uses problems that can be scheduled relatively easily to satisfy the hard constraint, and the focus is on optimising the soft one. Various methods have been applied to exam timetabling and a widely used set of problem data has been established by Carter et al. (1996). The problems are based on real data from thirteen universities and are often referred to as the University of Toronto benchmark data.

Recently, Bader-El-Den et al. (2009) have applied a grammar-based GP hyper-heuristic approach (see Section 2.5) to timetabling and have evaluated the method on the above mentioned data sets. As follows from using a hyper-heuristic approach, they compose several simple hand-crafted heuristics using conditional branching, to create a new heuristic. The grammar-based GP part of their method (called GPHH, Grammar-based Genetic Programming Hyper-Heuristic) claims to be "a new, hybrid way of using grammars in GP" different from both Koza's GP and Whigham's CFG-GP: the method operates according to the grammar directly on GP trees instead of derivation trees in order to increase performance. Other than this difference, which could be considered an implementation detail, the method based on the "grow" initialisation and the two familiar operators seems practically identical to CFG-GP.

Bader-El-Den et al. (2009) have evaluated their method on ten problems from the Toronto data set and their results were comparable with several existing algorithms. In the previous sections, we have applied GE and CFG-GP to several artificial problems that are either commonly used in the realm of genetic programming or highlight the use of grammar for ensuring type constraints. These artificial problems were also well-suited for comparing the characteristics of GE and CFG-GP. Exam timetabling is an opportunity to evaluate the performance of grammar-based GP methods on a real-world problem to which many methods other than GP have been applied. Another interesting difference from the previous experiments is that there are no known target solutions, instead there are many correct solutions that differ in how well they satisfy the soft constraints.

We will try to replicate the published results with CFG-GP, compare them with a similar setup of grammatical evolution, and we will also briefly discuss the performance in terms of computational time.

Let's start by stating the exam timetabling problem formally using the same notation as Bader-El-Den et al. (2009): Let E be the set of exams, S the set of students, P the set of time slots, $R : S \to \mathcal{P}(E)$ a registration function ($\mathcal{P}$ denotes a power set). The quadruple consisting of E, S, P, and R is a timetabling problem. A solution to the timetabling problem (E, S, P, R) is a function $O : E \to P$. The

solution is incorrect if $\exists\, s \in S, i, j \in E : i \neq j \,\&\, \{i, j\} \subseteq R(s) \,\&\, P(i) = P(j)$ (a student is to attend two exams at the same time), and the solution is incomplete if O is a partial function.

The soft constraint can be expressed quantitatively for correct solutions as a penalty p:

$$p = \frac{1}{S} \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} w(|p_i - p_j|) \cdot a_{ij} \tag{1}$$

where S is the total number of students, N is the total number of exams, a_{ij} is the number of students attending both exams i and j, p_i is the time slot to which exam i is assigned, and $w(d)$ is defined as 2^{5-d} if $d \leq 5$ and as 0 otherwise. This formula proposed by Carter et al. (1996) has since been used in the literature to compare the quality of solutions.

We will use a fitness value f based on the penalty formula:

$$f = \left(\sum_{i=1}^{M-1} \sum_{j=i+1}^{M} w(|p_i - p_j|) \cdot a_{ij} \right) + \alpha \cdot (N - M) \tag{2}$$

where M is the number of assigned exams. This is only a minor modification of the fitness function used by Bader-El-Den et al. (2009): we leave out the $\frac{1}{S}$ factor as it does not serve any purpose when computing fitness. The difference from the penalty function is that it allows for incomplete solutions. Bader-El-Den et al. (2009) do not mention any value for the α constant, but I have found out that any value large enough to disqualify incomplete solutions in tournaments with complete solutions works well. If we were to use a fitness-proportionate selection, however, it might not be the case.

Algorithms for timetabling that use heuristics usually proceed by repeatedly using the heuristic to select an exam and then its slot. It may happen that no slot is available for a given exam (the partial assignment would be incorrect), in which case the algorithm may stop, continue constructing an incomplete solution, or continue constructing an incorrect solution. This first phase may then be followed by a repair or improvement phase. Bader-El-Den et al. (2009) make a distinction between "constructive" algorithms without a repair or improvement phase and "improvement" algorithms, and they compare their results primarily with other constructive algorithms. They do not describe precisely the algorithm they use, but it is conceivably the one that continues to construct an incomplete solution even if some exams cannot be scheduled, as outlined in this pseudocode:

```
TIME-TABLING(E, S, P, R)
    ▷ T will contain all data structures for the heuristics and the (incomplete) assignment.
    T ← TT-SET-UP(E, S, P, R)
    while E ≠ ∅
        do e ← EXAM-HEURISTIC(T, E)
           s ← SLOT-HEURISTIC(T, e)
           if s ≠ NOT-FOUND
             then TT-ASSIGN(T, e, s)
           E ← E ∖ {e}                    ▷ whether have found a slot for e or not
    return TT-GET-ASSIGMENT(T)
```

We will therefore use this algorithm skeleton to evaluate our candidate solutions. The procedures TT-Set-Up and TT-Assign implement operations with data structures that serve for computations performed by the heuristics and maintain the incomplete assignment, which can be retrieved using TT-Get-Assigment. The evaluated hyper-heuristic is performed by calls to Exam-Heuristic and Slot-Heuristic. We will compose the hyper-heuristics from the same components as Bader-El-Den et al. (2009):

- branching based on current assignment size: V-Small, Small, Mid, Large;
- probabilistic random branching: Pr-20, Pr-40, Pr-50, Pr-70, Pr-90;
- the list of all unprocessed exams (E in the pseudocode): All-Exams;
- the list of all time slots: All-Slots,
- hand-crafted heuristics for selecting exams that act as list filters and can thus be pipelined:
  - Max-Conflict (largest degree), selects exams with most conflicts,
  - Least-Slot (saturation degree), selects exams with the least number of available slots,
  - Max-Students (largest enrolment), selects exams with the largest number of registered students;
- analogously working hand-crafted time slot heuristics:
  - Least-Cost, selects slots that increases the penalty the least,
  - Least-Busy, selects slots with the least number of assigned exams,
  - Most-Busy, selects slots with the largest number of assigned exams,
  - Least-Blocking, selects slots to which the least other conflicting unscheduled exams could be assigned;
- procedures for selecting a single exam or a slot from a list (Random-Exam, First-Exam,Random-Slot, First-Slot).

The exam selection heuristics are inspired by the graph-colouring problem (hence their alternative names) and were already proposed by Carter et al.

(1996). The time selection heuristics seem to have been introduced by Bader-El-Den et al. (2009) who give commonsense explanations for them. The basic idea is that even if some of the heuristics would not be useful alone, they may become useful in conjunction with each other or at particular stages of the timetable construction (using assignment size conditions).

The basic heuristics and other components can be combined according to the grammar shown Listing 3.10. Note particularly that conditional branching can occur anywhere, and so the pipelines of heuristics may form branches of arbitrarily shaped trees.

```
<assign> ::= while <exam>~=0 do <slot> end
<exam>   ::= randomExam(<eList>) | firstExam(<eList>)
<eList>  ::= maxConflict(<eList>)
          |  leastSlot(<eList>)
          |  maxStudents(<eList>)
          |  allExams(t)
          |  (<cond> and <eList> or <eList>)
<slot>   ::= randomSlot(<sList>) | firstSlot(<sList>)
<sList>  ::= leastCost(<sList>)
          |  leastBusy(<sList>)
          |  mostBusy(<sList>)
          |  leastBlocking(<sList>)
          |  allSlots(t)
          |  (<cond> and <sList> or <sList>)
<cond>   ::= <prob> | <size>
<size>   ::= vSmall(t) | small(t) | mid(t) | large(t)
<prob>   ::= pr20(t) | pr40(t) | pr50(t) | pr70(t) | pr90(t)
```

Listing 3.10: Context-free grammar in BNF for a timetabling hyper-heuristic in Lua. The `(··· and ··· or ···)` construct is a Lua idiom for a functional if-then-else condition with short-circuit (lazy) evaluation. The exam heuristic functions are implemented so that they return 0 when there are no exams left, the slot heuristic functions perform an exam-slot assignment if a slot is found. All data structures and the currently selected exam and slot are kept in the `t` data structure. This is a Lua equivalent of the grammar presented by Bader-El-Den et al. (2009).

3.9.1 Experimental setup

In addition to the grammar equivalent to the one used by Bader-El-Den et al. (2009), we will also report results with an alternative grammar without random selection of exams and slots and conditional branching for slots.

For CFG-GP, we will use the parameters specified by Bader-El-Den et al. (2009). The values of maximum initialisation height and maximum height are missing in their article as well as the already discussed α parameter of the fitness

function. As we do not have any beforehand knowledge of suitable tree height parameters, we will out of cautiousness use a low maximum initialisation tree height, letting the initialisation procedure raise it as needed for generation of unique individuals, and a high maximum tree height. There are only a few minor differences from the setup used by Bader-El-Den et al. (2009):

– We do not use mutually exclusive operators.
– We will evaluate individuals that use random selection or probabilistic conditions three times with different random number generator states and assign them the mean fitness value of the three runs. We will also report only this mean value in the results. Bader-El-Den et al. (2009) ran "the best performing individuals [. . .] for an extra 2 times". It is not clear if this concerns only the best individuals in a generation and whether the mean or the minimum value is used.
– We will perform only runs with 50 generations and 50 individuals in the population. In addition to this, Bader-El-Den et al. (2009) performed runs with "larger populations, with sizes ranging between 500 and 1000 individuals", but they do not give precise value for each experiment.

We will, as usual, use a lower nominal mutation rate for GE to reach an effective mutation rate similar to that of CFG-GP. The full list of parameters is provided in Table 3.9.)

Objective:	Find an exam and time slot selection heuristic for Time-Tabling.
Terminal operands:	All-Exams, All-Slots, V-Small, Small, Mid, Large, Pr-20, Pr-40, Pr-50, Pr-70, Pr-90.
Terminal operators:	Random-Exam*, First-Exam, Max-Conflict, Least-Slot, Max-Students, Exam-If-Then-Else, Random-Slot*, First-Slot, Least-Cost, Least-Busy, Most-Busy, Least-Blocking, Slot-If-Then-Else*. Alternatively, without starred items.
Grammar:	See Listing 3.10. Alternatively, without productions containing starred terminals.
Fitness cases:	One of the standard data sets `car91`, `car92`, `ear83`, `hec92`, `kfu93`, `lse91`, `sta83`, `tre92`, `uta92`, `yor83` (Carter et al., 1996). Candidates that use random selection or probabilistic conditions are evaluated three times with different random number generator states, and the mean fitness value is computed. The RNG state for the first evaluation is fixed for each run.
Raw fitness:	Formula (2) where $\alpha = 100000$.
Scaled fitness:	Same as raw fitness.
Algorithm:	Simple, generations: 50, population: 50.
Selection:	Tournament, size: 5.
CFG-GP initialisation:	"Grow" method (Whigham, 1995), maximum height: 4.
GE initialisation:	"Grow" method (Whigham, 1995), maximum height: 4.
GE operators:	Fixed-length one-point crossover, probability: 0.8. Bit-level mutation, probability: 0.01.
CFG-GP operators:	Crossover (Whigham, 1995), probability: 0.8. Mutation (Whigham, 1995), probability: 0.1.
Common parameters:	Maximum tree height: 20.
GE parameters:	Maximum wraps: 3. Codon size: 8.
Success predicate:	None.

Table 3.9: Exam timetabling parameters, CFG-GP and GE.

3.9.2 Results

In Table 3.10 we report the best results from ten runs for each method, grammar, and data set, as Bader-El-Den et al. (2009) did, along with the results reported by them. We use the same ten data sets.[4]

	car91	car92	ear83	hec92	kfu93	lse91	sta83	tre92	uta92	yor83
GPHH	‡5.12	4.46	‡37.10	‡11.78	‡14.72	‡11.11	‡158.70	†8.62	3.47	†40.56
CFG-GP	†5.11	‡4.44	38.78	12.67	15.21	11.93	159.44	8.78	3.49	40.73
GE	5.15	‡4.44	39.95	12.20	14.86	12.04	159.37	‡8.63	3.51	‡40.60
CFG-GP*	5.13	†4.37	37.29	12.15	14.74	11.75	160.22	‡8.63	‡3.44	40.94
GE*	5.22	4.51	39.68	12.27	15.23	11.96	160.22	8.79	‡3.44	41.47
Best other	Fuzzy 5.20	Fuzzy 4.52	Car. †36.4	Car. †10.8	Car. †14.0	Car. †10.5	Tabu †158.19	Fuzzy 8.67	TM †3.04	Fuzzy 40.66

*: alternative grammar. Best other: best other constructive heuristic, as reported by Bader-El-Den et al. (2009). †, ‡: best and second best results. Car.: Carter et al. (1996). Fuzzy: Asmuni et al. (2005). Tabu, TM (Tabu-Multi-stage): Burke et al. (2007).

Table 3.10: Penalties achieved by time tabling heuristics. Mean values are reported for stochastic heuristics evolved with CFG-GP and GE.

CFG-GP with the original grammar has in two cases (`car91`, `car92`) outperformed GPHH, which is the more interesting as these are two of the four cases in which Bader-El-Den et al. (2009) reported that GPHH already outperformed the best other constructive method.

GE with the original grammar has performed similarly to CFG-GP (four times better, five times worse). With the `car92` data set it has performed equally with CFG-GP, also beating both GPHH and the best other constructive method. With `car92` and `tre92` it has still performed better than the best other method, but worse than GPHH.

Because the grammar used by Bader-El-Den et al. (2009) is quite rich and the authors do not discuss their choices in much detail, I have experimented with removing various features. Most of the changes, such as permitting conditional branching only at the top level, or reducing the number of conditions, resulted in inferior performance. Removal of random selection of exams and slots and conditional branching for slots did, however, improve CFG-GP and GE results for seven and four data sets, respectively. CFG-GP with the alternative grammar outperformed all other methods in `car92`. Both CFG-GP and GE with the alternative grammar have outperformed GPHH. What is interesting about these results is that we have managed to remove a large part of randomness from the heuristics without impacting the quality of solutions.

[4] Bader-El-Den et al. (2009) refer to one of the Toronto data sets as `uta93`. Although they give parameters for `ute92`, it must be `uta92`, judging from the achieved penalty.

3.9.3 Computational time

Bader-El-Den et al. (2009) report approximate run times "on an ordinary PC", and their article was published in 2009. I performed my experiments on a consumer laptop manufactured in late 2008 (2.4 GHz Intel Core 2 Duo Apple MacBook with 2 GB of RAM), so the times should be comparable. According to Bader-El-Den et al. (2009), a run of GPHH took from 10 minutes to about 4 hours depending on the problem. With the same grammar and evaluating all individuals with random elements three times, all runs took me from less than a second to about 80 seconds. Even if they by mistake reported times for ten runs instead of one run, the running times of CFG-GP and GE using AGE are still an order of magnitude lower. Let's suppose that they meant 10 runs and compare the total running times for all experiments (10 runs for each data set) in Table 3.11.

	GPHH	CFG-GP	GE	CFG-GP*	GE*
CPU time	$\geq$ 12 hours	27 min 27 s	28 min 44 s	11 min 31 s	14 min 4 s

*: alternative grammar. Total CPU time (for both processor cores) is reported.

Table 3.11: Penalties achieved by time tabling heuristics.

Of course, the run times are only a matter of implementation, but the large performance gap is worth noting because the authors of GPHH regard their system as an improvement over CFG-GP specifically because it is more efficient by not being implemented using derivation trees. The alternative grammar has brought further substantial improvements because fewer individuals used random elements and thus fewer evaluations were necessary.

3.9.4 Conclusion

In this section we have compared CFG-GP and GE in terms of both quality of results and computational time with GPHH, an existing grammar-based GP systems with published results that are comparable to (and in some cases better than) those of other constructive methods for timetabling. In the case of CFG-GP, we could have expected to achieve similar results because GPHH is essentially based on the same method. In the case of GE, however, the results are one of the scarce examples of GE being compared with previously published results of methods that have performed well in some application. It is a pleasant discovery that GE has performed on a par with CFG-GP in evolving timetabling heuristics. I have not been able to find such comparison for any other problem field in the literature.

On two data sets CFG-GP has even slightly outperformed GPHH achieving the best results among the known constructive heuristics. GE has managed to do so on one data set. We have also shown that comparable results could be achieved in a substantially shorter run time using an efficient implementation and by adjusting the grammar. The shorter run times would be of critical

importance if the grammar-based GP methods were to be used in a more complex setup to evolve improvement heuristics for timetabling and compete with other improvement methods as Bader-El-Den et al. (2009) suggest for further research.

3.10 Conclusion

We have compared performance of GE and CFG-GP in a variety of application and discovered links between the differences in the produced tree shapes and the relative success of the two methods. The two methods performed similarly well a few times but in several applications the performance of GE was substantially worse.

In Section 3.4 we have found out that the more successful CFG-GP produces higher trees of more varied shapes, thus searching a larger space. In Section 3.5 we have made the same observation and verified it with different initialisation techniques. In the long run, both methods tended to converge to tree characteristics typical for them regardless of tree characteristics of the initial populations. In the two experiments done in Section 3.6 and Section 3.7, which compared performance with LOGENPRO, all three methods performed similarly. Nonetheless, we could observe the same patterns in development of tree characteristics for GE and CFG-GP. We have explained why the good results of GE were likely caused by its tendency to preserve low tree height throughout its run, the same property that would often have the opposite effect. In Section 3.8, we have shown that both GE and CFG-GP can use ADFs successfully but that the technique adapted from tree-based GP is not as effective for GE. As we have not used the usual mutation operators, the patterns of tree characteristics were different than in the earlier experiments but CFG-GP still produced higher and larger trees than GE, and was more successful even in the ADF-less versions of the experiments. GE has, however, found solutions of similar quality to those of CFG-GP in the real-world application to timetabling in Section 3.9.

We have thus shown a consistent difference between the tree shapes produced by GE and CFG-GP. GE tends to produce shorter trees, usually also less bushy and of less varied height and bushiness. The search of this smaller search space tends to be explorative thanks to the disruptive operators used in GE, and GE can provide slightly but significantly better results in some cases. In other cases, though, this seems to hamper performance. Additionally, the fact that the operators used in GE are grammar-agnostic clearly worsens performance with ADFs, and it would likely have the same effect with any grammar specifying subtrees with terminals of mutually incompatible types. We have not found GE to provide a substantial advantage in any of the problems.

In Section 3.9 we have also compared computational times needed by GE, CFG-GP, and another implementation of a CFG-GP-like algorithm to evolve timetabling heuristics. The efficient implementation of CFG-GP and GE I have

developed was at least an order of magnitude faster than the one used by Bader-El-Den et al. (2009). CFG-GP has performed slightly better than GE in terms of CPU time. This is a consequence of what individuals evolved during the run, and it demonstrates that the simpler operators of GE do not imply shorter overall run times.

Chapter 4

Implementation Notes

All experiments with GE and CFG-GP in Chapter 3 were done using an open-source framework called AGE (Algorithms for Grammar-based Evolution). I have created the initial version of the framework as an implementation of GE and general evolutionary algorithm elements as part of my bachelor thesis with the following goals:
– a clean, comprehensive implementation of standard algorithms,
– modularity,
– adequate documentation,
– versatile output,
– reproducible results,
– acceptable performance.

Thanks to the modular approach it was easy to extend the framework with implementation of CFG-GP and several new features. Although the framework aimed only at an "acceptable performance" in terms of computational time, it outperformed GEVA (an implementation of GE maintained at NCRA at University College Dublin) by an order of magnitude in benchmarks done as part of my bachelor thesis (Nohejl, 2009). The comparison of the CFG-GP implementation, which is now part of AGE, with published results from the CFG-GP-based GPHH framework (Bader-El-Den et al., 2009) was similarly favourable (Section 3.9.3).

In this chapter, we will describe the implementation of CFG-GP used in AGE (in Section 4.1), provide information about the accompanying files and about how to use them to replicate the results presented in the previous chapter (Section 4.2). User and developer documentation of the framework and tools, written as part of my bachelor thesis, is available inside the software package.

4.1 Implementation of CFG-GP

The original version of AGE already featured an efficient implementation of grammatical evolution. This implementation does not store derivation trees, instead phenotype strings are created directly from the codon strings (genotype), and when initialising individuals, genotype is generated directly. Thus deriva-

tion trees are used only implicitly without having to unnecessarily manipulate tree data structures. Both genotype and phenotype is stored in continuous arrays, and the implementation tries to prevent unnecessary allocations and deallocations of storage.

CFG-GP as described by (Whigham, 1995) represents individuals as trees, so that they can be manipulated by the tree-based operators. Tree data structures are traditionally stored as separately allocated nodes that contain pointers to their children. Such an approach is natural in most programming languages and offers the most flexibility when manipulating trees. When both the nodes and their trees are relatively small and the most frequent operations are sequential reads and writes of whole trees or large subtrees, a different representation is more efficient. Let's call it a *serialised tree*.

Nodes of a serialised tree are stored in depth-first left-to-right order, which is the same as the leftmost derivation order in the case of a derivation tree. This way nodes of each subtree are stored in a continuous sequence. Each node must contain (implicitly or explicitly) the number of its children in order to express structure of the tree, pointers to children can then be omitted and relations between nodes can be determined solely from their relative positions in the sequence. Size of each subtree of a serialised tree is optionally stored in the node where the subtree is rooted. These sizes are not strictly necessary to manipulate a serialised trees but they make it possible to replace a subtree without having to retrieve numbers of children from all its nodes. An example of a serialised tree is shown in Figure 4.1.

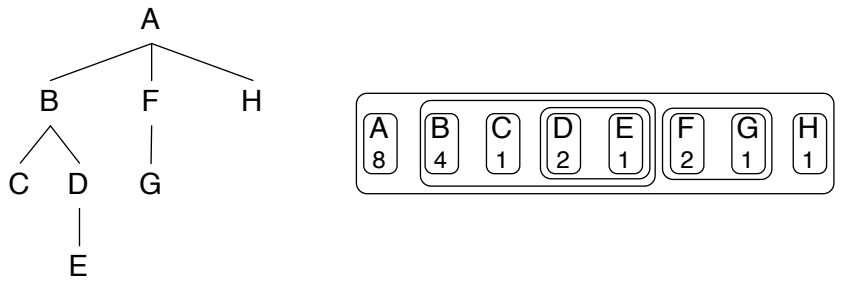


Figure 4.1: A tree in the usual representation and its corresponding serialised tree with subtree sizes in each node. Subtree boundaries are outlined.

One approach to representing CFG-GP derivation trees is therefore to represent them as serialised trees. Let's consider what information would we need to have about the nodes in order to implement the CFG-GP operators. If we know which production is used in each nonterminal node, we can also tell the number of its children and their labels (terminal and nonterminal symbols). Mutation and crossover sites are always located on nonterminal nodes (Whigham, 1995), thus terminal nodes do not need to be stored and we only need to store information about which productions were used at nonterminal nodes. Let's number the productions for each nonterminal from zero, and call these numbers *choices*. It may happen that some nonterminals have only one

production, offering a single choice (effectively no choice), in that case the node does not need to be stored in the serialised representation.[1]

A minimal representation of a derivation tree is therefore a serialised tree of its choice nodes. Our definition of choice nodes (see page 28) coincides with nonterminal nodes with more than one choice, and as we have noted, these are precisely the nodes encoded by the codons used in grammatical evolution. The order in which the nodes are stored is also the same. Thus a serialised choice-node tree is a special case of a GE chromosome that represents the same derivation. This correspondence is illustrated by Figure 4.2.

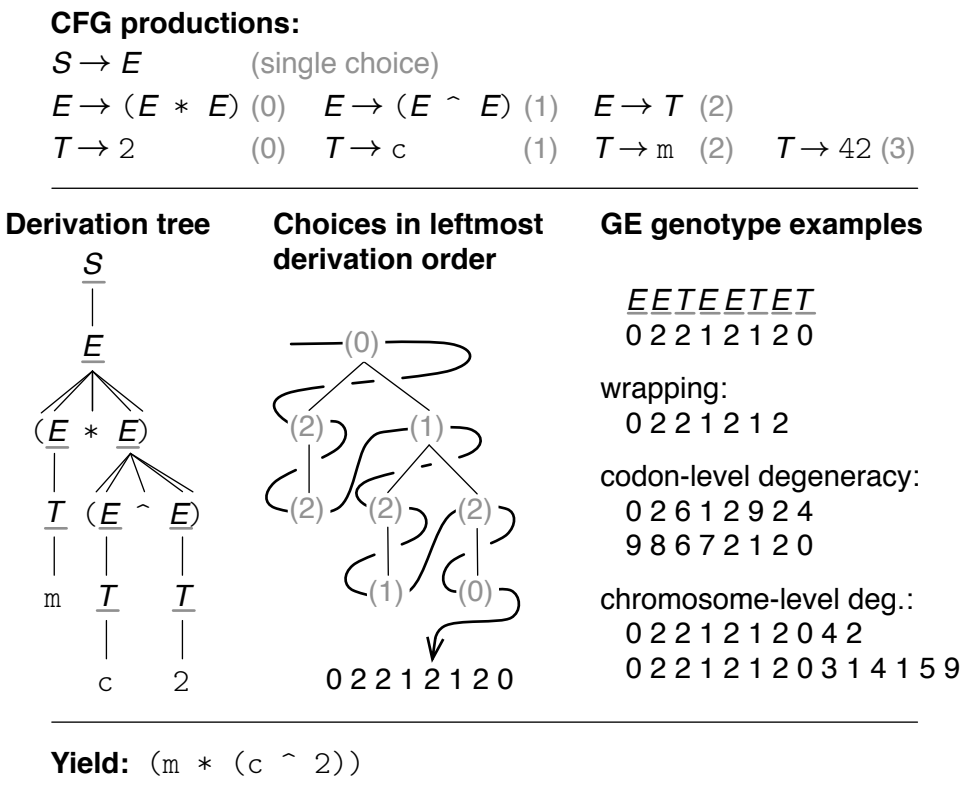


Figure 4.2: Three representations of the derivation of the string `(m * (c ^ 2))` according to a set of CFG productions: (1) derivation tree: nonterminal nodes, at which crossover or mutation can occur in CFG-GP, are underlined; (2) choices in leftmost derivation order: basis for genotype in GE, and for the implementation of CFG-GP in AGE; (3) GE genotype: codon values may correspond to choices exactly, codons may be reused in a circular fashion (wrapping), codons are interpreted modulo number of choices (codon-level degeneracy), unused codons may be appended (chromosome-level degeneracy).

The implementation of CFG-GP in AGE is based on serialised trees of choice nodes. Individuals represented this way can share evaluation procedures, in-

[1] If we were to follow (Whigham, 1995) literally, we would have to store all nonterminal nodes as crossover or mutation can occur on any one of them. Crossover or mutation on a node with single choice is, however, equivalent to crossover or mutation on its parent node (except for a single-choice root node). Additionally, single-choice nodes can be considered a degenerate cases, which bias operators towards their parents.

cluding the genotype-phenotype mapping, with GE. The tree-based initialisation procedures used for GE need only one modification: removal of genetic code degeneracy (see Figure 4.2 for comparison). Other general evolutionary algorithm elements such as selection methods can be shared regardless of representation. Only the mutation and crossover operators need to be implemented separately.

For the purposes of these operators, AGE generates other node information such as the already mentioned subtree size. Because such data can be generated quickly, it is relatively small, and its reuse is limited, it is actually more efficient to re-generate it every time it is needed. Along with this data, statistics about tree characteristics that we have used in Chapter 3 can be generated using common procedures for both GE and CFG-GP.

The main advantages of this implementation is that CFG-GP trees are stored in compact and easily manipulated data structures (arrays of choice numbers and temporarily generated arrays of other node data). Note that in contrast to traditional Lisp-based GP the derivation trees to which operators are applied cannot share the representation with the parse trees used when evaluating individuals because arbitrary grammars are allowed. We cannot avoid generating the full string representation of each individual (the phenotype in GE) which is then parsed in order to be evaluated. Therefore the minimal representation is advantageous.

4.2 Accompanying files

This thesis is accompanied by the following files on the enclosed medium:

- `Adam-Nohejl-2011-Master-Thesis.pdf`, a PDF version of this thesis,
- `AGE-1.1.tar.gz`, the AGE source package with documentation,
- `Experiments.tar.gz`, results from LOGENPRO and several scripts:
  - `AGE-run.sh`, a shell script to run experiments for this thesis,
  - `AGE-stats.sh`, a helper R script for `stats.sh`,
  - `exstats.sh`, a helper shell script for `stats.sh`,
  - `plot.r`, an R script to draw plots used in this thesis,
  - `runner.sh`, a helper shell script for `AGE-run.sh`,
  - `stats.sh`, a shell script to compute statistics for the plots.

The included version of the AGE framework has been extended mainly by the CFG-GP algorithm elements as described above. The implementations of all experimental problems used in this thesis are also part of the default build. The software is portable: it can be built on a POSIX-compliant system (tested on Mac OS X, NetBSD and Linux) with a decent standards-compliant

C/C++ compiler (tested with GCC and Clang) and it does not have any external dependencies (details in the included documentation). The whole package is in the archive named `AGE-1.1.tar.gz` and has a customary UNIX source package structure (you can start by reading the `README` and `INSTALL` text files).

To replicate the experiments done in Chapter 3, follow these steps:

1. Build and install AGE from the source package and make sure it is in your search path (your `PATH` variable must include the directory where you have installed the `AGE` executable). Suggested commands for installation into your home directory:

   ```
   % tar xzf path/AGE-1.1.tar.gz
   % cd AGE-1.1; make INSTALL_PREFIX=~ install
   % cd ..; export PATH=~/bin:"$PATH"
   ```

   where *path* is path to the directory containing the file `AGE-1.1.tar.gz` (mount point of the accompanying disc).

2. Expand the `Experiments.tar.gz` archive. The resulting `Experiments` directory will contain the results obtained from LOGENPRO that we have presented in Chapter 3. Suggested command:

   ```
   % tar xzf path/Experiments.tar.gz
   ```

   where *path* is path to the directory containing the file `Experiments.tar.gz`.

3. Change your working directory to the newly created Experiments directory. Suggested command:

   ```
   % cd Experiments
   ```

4. Run the `AGE-run.sh` script. Running all the experiments will likely take a few hours to finish. Alternatively, you can open the file in a text editor and extract only commands for the experiments you want to run or adjust options. Suggested command:

   ```
   % ./AGE-run.sh
   ```

   Output from the experiments is in a documented format and results from each run can be displayed using a XSLT style sheet (consult the AGE documentation for details).

5. Optionally, to compute the necessary statistics and render the plots used in the thesis, run the `stats.sh` script and then the `plot.r` script. (The R statistical package must be installed.) Suggested commands:

   ```
   % ./stats.sh; ./plot.r
   ```

The generated plots will be saved in PDF files with the following name patterns:

- `plot-ad*.pdf`: Symbolic regression with ADFs (Section 3.7),
- `plot-at*.pdf`: Santa Fe ant trail (Section 3.5),
- `plot-bp*.pdf`: Boolean parity functions with ADFs (Section 3.8),
- `plot-dp*.pdf`: Dot product symbolic regression (Section 3.6),
- `plot-sr*.pdf`: Simple symbolic regression (Section 3.4).

Conclusion

We have discussed the problems encountered by traditional genetic programming that can be solved using grammars. We have reviewed three different grammar-based GP methods, described their distinctive features, and tested them in a number of benchmark application, including the real-world application of finding heuristics for timetabling. We have implemented two of the methods, grammatical evolution and context-free grammar genetic programming, in a common framework, which allowed us to compare their performance in identical setups. In addition to comparing performance, we have focused on a statistical analysis of tree characteristics produced by each method, which has provided insight into differences between GE and CFG-GP. As most of the research in this field focuses either on novel applications or compares results only with traditional genetic programming, we have been able to shed more light on actual performance and comparative advantages of the grammar-based methods.

In Chapter 1 we have introduced the basic concepts of genetic programming and formal grammars, and we have pointed to the following problems encountered by GP that grammars can solve: adaptability to different programming languages and environments, the problem of closure and multiple types, and the problem of declarative representation of knowledge. We have presented three different approaches to applying grammars to GP: context-free grammar genetic programming, LOGENPRO, and grammatical evolution. We have described their distinctive features, and what they imply for performance of these methods. Based on this survey, we have focused on the role of the power of logic grammars in LOGENPRO, on the unclear effect of operators in GE, and the problem of bias (whether wanted or not) encoded in a grammar.

In Chapter 2 we have described several problem classes and areas of application, most of them admittedly artificial and of little practical value but useful for comparing the performance of grammar-based GP methods. We have also briefly presented hyper-heuristics, a field that offers possibilities of real-world application to grammar-based genetic programming.

In Chapter 3 we have tested GE and CFG-GP in six different applications. In two of them we have also compared the results with LOGENPRO, and found no particular advantage in using logic grammars in the applications that Wong and Leung (2000) used to demonstrate the abilities of LOGENPRO. In one application we have compared the performance of CFG-GP and GE with tree-based GP (Koza, 1992, results reported by), demonstrated that ADFs can

be carried over to the grammar-based methods, particularly to CFG-GP, and explained the inferior performance of GE. In the last application we have compared performance both in terms of quality and speed with recently published results of a grammar-based hyper-heuristic framework (GPHH, Bader-El-Den et al., 2009) for timetabling. The comparison was very favourable: using CFG-GP we have outperformed the published results in two problem instances, in which GPHH was already reported to be the best constructive heuristic to date. Additionally, our implementation is at least an order of magnitude faster. I believe that such a large improvement in speed could open new possibilities for practical applications of grammar-based GP methods.

Perhaps even more important output from the six experiments is that GE does not provide a substantial advantage in any of them. In several experiments it performs substantially worse, and we have been able to link this performance to the tree characteristics resulting from its operators, and the fact that grammatical constraints in GE are enforced only by the genotype-phenotype mapping, not by the operators. Additionally, we have demonstrated that the simpler operators of GE do not transform in shorter overall run times. I see this as the largest contribution of my thesis, as the effect of operators used in GE is hard to analyse, and I have not been able to find such comparisons with other grammar-based GP methods in the literature.

The thesis has fulfilled its goal of thoroughly comparing the grammar-based GP methods, has showed that CFG-GP, the simplest of the three compared methods, provides comparable or even better results than the other two, and has demonstrated that CFG-GP can be implemented very efficiently. We have several times touched the problem of embedding knowledge (bias) in grammars. The choice of grammar obviously has a great effect on performance of any grammar-based GP algorithm, and if at least part of the problem could be automated by co-evolving grammars or transforming them into more suitable forms, it would provide great advantage, especially when CFG-GP is applied to more complex problems. This seems to be the most promising area for further research.

Bibliography

Hishammuddin Asmuni, Edmund Burke, Jonathan Garibaldi, and Barry McCollum. Fuzzy multiple heuristic orderings for examination timetabling. In Edmund Burke and Michael Trick, editors, *Practice and Theory of Automated Timetabling V*, volume 3616 of *Lecture Notes in Computer Science*, pages 334–353. Springer Berlin / Heidelberg, 2005.

Mohamed Bahy Bader-El-Den, Riccardo Poli, and Shaheen Fatima. Evolving timetabling heuristics using a grammar-based genetic programming hyper-heuristic framework. *Memetic Computing*, 1(3):205–219, 2009.

Edmund K Burke, Barry Mccollum, Amnon Meisels, Sanja Petrovic, and Rong Qu. A graph-based hyper-heuristic for educational timetabling problems. *European Journal of Operational Research*, 176:177–192, 2007.

Michael W. Carter, Gilbert Laporte, and Sau Yan Lee. Examination timetabling: Algorithmic strategies and applications. *J Oper Res Soc*, 47(3):373–383, 03 1996. URL `http://dx.doi.org/10.1057/jors.1996.37`.

William W. Cohen. Compiling prior knowledge into an explicit bias. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 102–110. Morgan Kaufmann, 1992.

William W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artif. Intell.*, 68(2):303–366, 1994.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

Ian Dempsey, Michael O'Neill, and Anthony Brabazon. *Foundations in Grammatical Evolution for Dynamic Environments*, volume 194 of *Studies in Computational Intelligence*. Springer, 2009. URL `http://www.springer.com/engineering/book/978-3-642-00313-4`.

David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1989. ISBN 0201157675.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2 edition, 2000. ISBN 0201441241.

John R. Koza. *Genetic programming: On the programming of computers by natural selection*. MIT Press, 1992. ISBN 0-262-11170-5.

Sean Luke. Genetic programming produced competitive soccer softbot teams for RoboCup97. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 214–222. Morgan Kaufmann, 1998.

Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. Co-evolving soccer softbot team coordination with genetic programming. In Hiroaki Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, pages 398–411. Springer-Verlag, 1997.

David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3:199–230, 1994.

Stephen Muggleton. Inductive logic programming: derivations, successes and shortcomings. *SIGART Bull.*, 5:5–11, January 1994. ISSN 0163-5719.

Peter Naur. Revised report on the algorithmic language Algol 60. *CACM*, 6(1):1–17, 1963. URL `http://www.cc.gatech.edu/data_files/classes/cs6390/readings/algol.pdf`.

Adam Nohejl. *Grammatical Evolution*. Bachelor thesis, available at `http://nohejl.name/age/documentation/`, 2009.

Michael O'Neill and Anthony Brabazon. Evolving a logo design using lindenmayer systems, postscript and grammatical evolution. In Jun Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, pages 3788–3794, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.

Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Springer, 1st edition, 2003.

Michael O'Neill, Erik Hemberg, Conor Gilligan, Eliott Bartley, James McDermott, and Anthony Brabazon. *GEVA – Grammatical Evolution in Java (v 1.0)*. Technical report, UCD School of Computer Science and Informatics, 2008. URL `http://www.csi.ucd.ie/files/ucd-csi-2008-09.pdf`.

Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. URL `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

Conor Ryan, J. J. Collins, and Michael O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 83–95, Paris, 14-15 April 1998. Springer-Verlag. ISBN 3-540-64360-5.

C. M. Sperberg-McQueen. *A brief introduction to definite clause grammars and definite clause translation grammars*. A working paper prepared for the W3C XML Schema Working Group, 2004. URL `http://cmsmcq.com/2004/lgintro.html`.

Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 2nd edition, 1994. ISBN 978-0-262-69163-5.

Peter Whigham. Inductive bias and genetic programming. In *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, pages 461–466, 1995.

Peter Whigham. Search bias, language bias and genetic programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 230–237. MIT Press, 1996.

Man Leung Wong and Kwong Sak Leung. Applying logic grammars to induce sub-functions in genetic programming. In *1995 IEEE Conference on Evolutionary Computation*, volume 2, pages 737–740, Perth, Australia, 29 November - 1 December 1995. IEEE Press.

Man Leung Wong and Kwong Sak Leung. *Data Mining Using Grammar-Based Genetic Programming and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 079237746X.

List of Abbreviations

ADF automatically defined function. 9, 10, 15, 17, 18, 23, 24, 54–58, 61–64, 75, 83

BNF Backus-Naur form. 5

CFG context-free grammar. 4–7, 12, 13, 15, 54

CFG-GP context-free grammar genetic programming. 12, 14, 15, 17, 18, 21, 25, 27–31, 34–36, 38–40, 43, 44, 47–52, 54–58, 61–64, 67, 70–80, 83, 84

CV coefficient of variance. 28, 31, 35, 43, 44, 52

DCG definite clause grammar. 6, 7, 11, 14, 15, 17, 47

GA genetic algorithms. 3, 9, 16, 17, 22

GE grammatical evolution. 15–18, 21, 22, 25, 27–31, 34–36, 38–40, 43, 44, 47–52, 54–58, 61–64, 67, 71–77, 79, 80, 83, 84

GP genetic programming. 1, 8–12, 14–17, 21–25, 48, 51, 55, 61–64, 67, 80, 83

ILP inductive logic programming. 11, 12, 14, 17, 25

STGP strongly-typed genetic programming. 9, 11, 13, 15