On Linear Genetic Programming

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Universität Dortmund
am Fachbereich Informatik
von

Markus Brameier

Dortmund

Feb. 2004

Date of Submission: 28. May 2003
Date of Oral Examination: 13. February 2004

Dean: Prof. Dr. Bernhard Steffen

Referees:
Prof. Dr. Wolfgang Banzhaf
Prof. Dr. Martin Riedmiller
Prof. Dr. Peter Nordin (external)

To my parents

Abstract

The thesis is about linear genetic programming (LGP), a machine learning approach that evolves computer programs as sequences of imperative instructions. Two fundamental differences to the more common tree-based variant (TGP) may be identified. These are the graph-based functional structure of linear genetic programs, on the one hand, and the existence of *structurally* noneffective code, on the other hand.

The two major objectives of this work comprise (1) the development of more advanced methods and variation operators to produce better and more compact program solutions and (2) the analysis of general EA/GP phenomena in linear GP, including intron code, neutral variations, and code growth, among others.

First, we introduce efficient algorithms for extracting features of the imperative and functional structure of linear genetic programs. In doing so, especially the detection and elimination of noneffective code during runtime will turn out as a powerful tool to accelerate the time-consuming step of fitness evaluation in GP.

Variation operators are discussed systematically for the linear program representation. We will demonstrate that so called *effective instruction mutations* achieve the best performance in terms of solution quality. These mutations operate only on the (structurally) effective code and restrict the mutation step size to one instruction.

One possibility to further improve their performance is to explicitly increase the probability of neutral variations. As a second, more time-efficient alternative we explicitly control the mutation step size on the effective code (effective step size). Minimum steps do not allow more than one effective instruction to change its effectiveness status. That is, only a single node may be connected to or disconnected from the effective graph component. It is an interesting phenomenon that, to some extent, the effective code becomes more robust against destructions over the generations already implicitly.

A special concern of this thesis is to convince the reader that there are some serious arguments for using a linear representation. In a crossover-based comparison LGP has been found superior to TGP over a set of benchmark problems. Furthermore, linear solutions turned out to be more compact than tree solutions due to (1) multiple usage of subgraph results and (2) implicit parsimony pressure by structurally noneffective code.

The phenomenon of code growth is analyzed for different linear genetic operators. When applying instruction mutations exclusively almost only neutral variations may be held responsible for the emergence and propagation of intron code. It is noteworthy that linear genetic programs may not grow if all neutral variation effects are rejected *and* if the variation step size is minimum. For the same reasons effective instruction mutations realize an implicit complexity control in linear GP which reduces a possible negative effect of code growth to a minimum. Another noteworthy result in this context is that program size is strongly increased by crossover while it is hardly influenced by mutation even if step sizes are not explicitly restricted.

Finally, we investigate program teams as one possibility to increase the dimension of genetic programs. It will be demonstrated that much more powerful solutions may be found by teams than by individuals. Moreover, the complexity of team solutions remains surprisingly small compared to individual programs. Both is the result of specialization and cooperation of team members.

About the Author

In 1997 Markus Brameier received a diploma degree in computer science (subsidiary subject: theoretical medicine) from the University of Dortmund in Germany. Until 2002 he was a research associate in the group of Prof. Dr. Wolfgang Banzhaf at the Department of Computer Science of the same university, more precisely at the Chair of System Analysis and Evolutionary Algorithms held by Prof. Hans-Paul Schwefel. The author was a member of the Collaborative Research Center SFB 531 *Computational Intelligence* instituted in Dortmund. His main research focus is on genetic programming and bioinformatics. Other research interests include evolutionary algorithms in general, neural networks, and medical informatics.

Publications

Book Chapters

W. Banzhaf, M. Brameier, M. Stautner, and K. Weinert. *Genetic Programming and its Application in Machining Technology.* In H.-P. Schwefel, I. Wegener, and K. Weinert (eds.) *Advances in Computational Intelligence – Theory and Practice*, Springer, Berlin, 2002.

Journal Articles

M. Brameier and W. Banzhaf, *Evolving Teams of Predictors with Linear Genetic Programming.* Genetic Programming and Evolvable Machines, vol. 2(4), pp. 381–407, 2001.

M. Brameier and W. Banzhaf, *A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining.* IEEE Transactions on Evolutionary Computation, vol. 5(1), pp. 17–26, 2001.

P. Nordin, W. Banzhaf, and M. Brameier, *Evolution of a World Model for a Miniature Robot Using Genetic Programming.* Robotics and Autonomous Systems, vol. 25, pp. 105–116, 1998.

Refereed Conference Papers

M. Brameier and W. Banzhaf, *Neutral Variations Cause Bloat in Linear GP.* In C. Ryan et al. (eds.) *Proceedings of the Sixth European Conference on Genetic Programming* (EuroGP 2003), LNCS 2610, pp. 286–296, Springer, Berlin, 2003. (**Best Poster Paper Award**)

M. Brameier and W. Banzhaf, *Explicit Control of Diversity and Effective Variation Distance in Linear Genetic Programming.* In J.A. Foster et al. (eds.) *Proceedings of the Fifth European Conference on Genetic Programming* (EuroGP 2002), LNCS 2278, pp. 37–49, Springer, Berlin, 2002. (**Best Paper Award**)

M. Brameier, F. Hoffmann, P. Nordin, W. Banzhaf, and F. Francone, *Parallel Machine Code Genetic Programming.* In W. Banzhaf et al. (eds.) *Proceedings of the Genetic and Evolutionary Computation Conference* (GECCO '99), Morgan Kaufmann, San Francisco, CA, 1999.

P. Nordin, F. Hoffmann, F. Francone, M. Brameier, and W. Banzhaf, *AIM-GP and Parallelism.* In Proceedings of the Congress on Evolutionary Computation (CEC '99), pp. 1059–1066, IEEE Press, Piscataway, NJ, 1999.

W. Kantschik, P. Dittrich, M. Brameier, and W. Banzhaf, *Empirical Analysis of Different Levels of Meta-Evolution.* In *Proceedings of the Congress on Evolutionary Computation* (CEC '99), pp. 2086–2093, IEEE Press, Piscataway, NJ, 1999.

W. Kantschik, P. Dittrich, M. Brameier, and W. Banzhaf, *Meta-evolution in graph-GP.* In R. Poli et al. (eds.) *Proceedings of the Second European Conference on Genetic Programming* (EuroGP '99), pp. 15–28, Springer, Berlin, 1999.

Technical Reports

M. Brameier and W. Banzhaf, *Explicit Control Of Diversity and Effective Variation Distance in Linear Genetic Programming.* Technical Report CI-123/01, Collaborative Research Center 531, University of Dortmund, 2001. (extended version)

M. Brameier and W. Banzhaf, *Effective Linear Program Induction.* Technical Report CI-108/01, Collaborative Research Center 531, University of Dortmund, 2001.

M. Brameier, W. Kantschik, P. Dittrich, and W. Banzhaf, *SYSGP – A C++ Library of Different GP Variants.* Technical Report CI-98/48, Collaborative Research Center 531, University of Dortmund, 1998.

Acknowledgements

The author owes gratitude to his supervisor Prof Dr. Wolfgang Banzhaf for support, valuable advises, and great freedom in research. The author would also like to thank his former colleagues Dr. Peter Dittrich and Wolfgang Kantschik for many fruitful discussions. Financial support for this work was granted by the German Research Community DFG (Deutsche Forschungsgemeinschaft), Collaborative Research Center SFB (Sonderforschungsbereich) 531, project B2, and is gratefully acknowledged.

Markus Brameier
Dortmund, May 2003

Contents

Chapter 1

Introduction

Contents

1.1 Evolutionary Algorithms

Evolutionary algorithms (EA) mimic aspects of natural evolution to optimize a solution towards a predefined goal. Following Darwin's principle of natural selection, differential fitness advantages are exploited in a population to lead to better solutions. Different research subareas of evolutionary algorithms have emerged, such as *genetic algorithms* (GA), *evolution strategies* (ES), and *evolutionary programming* (EP). A comparatively young approach in this context is *genetic programming* (GP). Evolutionary algorithms as a whole together with neural networks and fuzzy logic are considered as disciplines of *computational intelligence* (CI) [92]. A general evolutionary algorithm may be summarized as follows:

ALGORITHM 1.1 (*general evolutionary algorithm*)

1. Randomly initialize a population of individual solutions.
2. Randomly select individuals from the population that are fitter than others by using a certain *selection method*. The *fitness* measure defines the problem the algorithm is expected to solve.
3. Generate new variants by applying the following *genetic operators* for certain probabilities:
   - □ *Reproduction*: Copy an individual without change.
   - □ *Recombination*: Exchange substructures between individuals.
   - □ *Mutation*: Randomly replace a single atomic unit in an individual.
4. Calulate the fitness of new individuals.
5. If the termination criterion is not met, $\rightarrow$ 2.
6. Stop. The best individual represents the best solution found.

While in genetic algorithms [40, 32] the individuals are represented as fixed-length binary strings, evolution strategies [75, 91] operate on real-valued vectors. Both techniques are applied primarily in parameter optimization. Compared to that genetic programming varies individuals on a more symbolic level as computer programs. That is, the representation is executable and usually of variable size and shape.

In a more general sense genetic programming may also be regarded as a method of *machine learning* (ML) which studies computer algorithms that learn by experience [61]. Especially some of the early machine learning approaches show clear resemblance to modern GP. Friedberg [30, 31] attempted to solve simple problems by teaching a computer to write computer programs. Due to his choice of search strategy, however, his results were limited. Later evolutionary programming [29] was introduced as a method that uses finite state automata as a representation of individuals. This innovative work could be argued to be the first successful evolutionary algorithm for automatic program induction. It was then Cramer [26] who first applied an EA to (more general) programs that were represented as variable-length tree structures. But only the detailed work of Koza [50, 51] could demonstrate the feasibility of this approach in well-known application areas. He also gave the field its name "genetic programming".

1.2 Genetic Programming

Genetic programming (GP) may be defined generally as any direct evolution or breeding of computer programs for the purpose of inductive learning. In particular, this definition is supposed to be independent of a special type of program representation. In principle, GP may solve the same range of problems as other machine learning techniques, like neural networks. Most of todays real-world applications of GP demonstrate its abilities in data mining, i.e., the discovery of regularities within large data domains. For supervised learning tasks that means to create predictive models, i.e., classifiers or approximators, that learn a set of known (labeled) data and generalize to a set of unknown (unlabeled) data. Other application areas of GP may comprise, for instance, control problems, time series prediction, signal processing and image processing.

Genetic programs may be regarded as prediction models that approximate an *objective function* $f : I^n \to O^m$ with I^n denotes the input data space of dimension n and O^m is the m-dimensional output data space. In most cases there is only $m = 1$ output. Genetic programs may also complete missing (unknown) parts of an existing model. Other evolutionary algorithms, like genetic algorithms or evolution strategies, minimize an existing objective function (model) by searching for the optimum setting of its variables (model parameters).

The objective function f itself represents the problem to be solved by GP. In practice this function is usually unknown and defined only incompletely by a relatively small set of input-output vectors $T = \{(\vec{i}, \vec{o}) \mid \vec{i} \in I' \subseteq I^n, \vec{o} \in O' \subseteq O^m, f(\vec{i}) = \vec{o}\}$. The evolutionary process searches for a program that represents the best solution to a given problem, i.e., that maps the given training set T best. Training examples are also referred to as *fitness cases* in GP. GP models are not only expected to predict the outputs of all training inputs I' most precisely but also many inputs from $I^n \backslash I'$. That is, the genetic programs are desired to generalize from the training data to unknown data. The generalization ability is verified by means of input-output examples from the same data domain as (but different from) the training examples.

The *genotype space* $\mathcal{G}$ in GP includes all programs of a certain *representation (type)* that can be composed of elements from a *programming language* $\mathcal{L}$. If we assume that programs do not induce side-effects the *phenotype space* $\mathcal{P}$ denotes the set of all mathematical functions $f_{gp} : I^n \to O^m$ with $f_{gp} \in \mathcal{P}$ that can be expressed by programs $gp \in \mathcal{G}$. The used programming language $\mathcal{L}$ is defined by the user over the *instruction set* (or *function set*) and the so called *terminal set*. The latter may comprise input values, constants, and memory variables.

The *fitness function* $\mathcal{F} : \mathcal{P} \to V$ measures the prediction quality, i.e., *fitness*, of a phenotype $f_{gp} \in \mathcal{P}$. For this thesis we assume the range of fitness values to be $V = \mathbb{R}_0^+$ for continuous problems and $V = \mathbb{N}_0$ for discrete problems. Usually fitness is derived from a mapping error between the predicted model f_{gp} and the desired model f. Since, in general, fitness cases represent a fraction of the problem data space only, fitness may reflect the phenotype behaviour of a program only in part.

Fitness evaluation of individuals is by far the most time-critical step of a GP algorithm since a genetic program has to be *executed* at least once for each fitness case in the fitness function. Prior to that, the genotype representation gp has to be translated into the phenotype function f_{gp}. Such a genotype-phenotype mapping is usually deterministic and produced by an *interpreter* $f_{int} : \mathcal{G} \to \mathcal{P}$ with $f_{int}(gp) = f_{gp}$ and $\mathcal{F}(gp) := \mathcal{F}(f_{gp})$. Both functions f_{int} and $\mathcal{F}$ are not bijective. That means a phenotype may be represented by more than one genotype and different phenotypes may have the same fitness.

The composition of the instruction set and the terminal set determines the expressiveness of the programming language $\mathcal{L}$. On the one hand, this language must be mighty enough to represent the optimum solution or at least a good suboptimum solutions. On the other hand, solution finding becomes more difficult if the *search space* of programs $\mathcal{G}$ is increased unnecessarily by too large sets of program components. If $\mathcal{L}$ is Turing-complete, every computable function may be found, in principle, provided that the maximum program size is sufficiently large to represent it. In practice it is recommended, however, to define the language as small as necessary. Genetic programming requires a certain knowledge here from the user about the problem domain to solve this trade-off situation. Another problem is that we cannot know beforehand if a program will terminate or not if we choose the representation such that the underlying language is Turing-complete. Since Turing-completeness requires infinite loops, a maximum time bound is necessary to guarantee a finite execution of program. One possibility is to restrict the maximum number of executed instructions.

There are many ways to represent a certain function by a program. This is mostly due to neutral code in genotypes that is not expressed in the phenotype. The complexity of a genetic program is usually measured as the number of instructions it holds. A growing variable-length representation is important in GP since, in general, the minimum representation size of the optimum solution is unknown. Following the *principle of Occam's Razor* among all solutions with equal fitness the shortest solution should be preferred. This solution is supposed to achieve the best generalization performance. In GP it depends on the expressiveness of the used programming language and on the variability of the representation form how compact a program is possible for a certain objective function.

The maximum size of programs has to be restricted in general to prevent programs from growing without bound and using up all system memory. If no maximum restriction is imposed on the representation size not only the generalization ability of solutions may be reduced, but also the efficiency of genetic operations. Additionally, the critical time for the fitness evaluation of programs is increased. A too small maximum complexity bound, on the other hand, may restrict a solution finding if it is not sufficient to represent the optimum solution. The user is asked again to find a good trade-off here. Both the success of the evolutionary search and the growth of programs depend not only on the representation but on the variation operators, too.

Let $P(t) \subset \mathcal{G}$ denote the state of a *population* at time (*generation*) t. From a random subpopulation $P' \subseteq P(t)$ of $n = |P'|$ individuals a *selection operator* $s : \mathcal{G}^n \times \mathcal{P}^n \rightarrow \mathcal{G}^\mu$ selects $\mu < n$ individuals for variation that show a better fitness than others. For global selection schemes $P' = P(t)$ is true. The selection operator determines among already visited search points (in genotype space and in phenotype space) from where the search may be continued most promisingly. Depending on a *reproduction rate* p_{rr} the μ parents are transfered over into population $P(t+1)$ of the next generation $t+1$.

A *genetic operator* or *variation operator* $v : \mathcal{G}^\mu \rightarrow \mathcal{G}^\lambda$ creates λ offsprings out of the μ selected parents from population $P(t)$. These λ new individuals become part of population $P(t+1)$, too. In other words, λ new search points are visited in the genotype search space. If $\mu < \lambda$ a parent produces more than one offspring. Usually recombination in GP creates two offsprings from two parents, i.e., $\mu = \lambda = 2$, while for mutations $\mu = \lambda = 1$ is used. All genetic operators must guarantee, first, that no syntactically incorrect programs are generated during evolution (*syntactic closure*). Second, the value and the type of each instruction argument must be from defined ranges (*semantic protection*). The calculation of a new search point is much less expensive than the fitness evaluation in GP and may be neglected, at least if the application of a variation operator does not take more than linear calculation time $O(n)$ with program size n.

1.3 Linear Genetic Programming

In recent years different variants of genetic programming have emerged all following the basic idea of GP, i.e., the automatic evolution of computer programs. Three basic forms of representation may be distinguished for genetic programs. Besides the traditional tree representations these include linear and graph representations [11].

The tree programs used in Koza-style genetic programming correspond to expressions from a functional programming language. This classic approach is also referred to as *tree-based genetic programming* (TGP). Functions are located at inner nodes while the leafs hold input values or constants. In contrast, *linear genetic programming* (LGP) denotes a GP variant that evolves sequences of instructions from an imperative programming language or machine language. For this thesis instructions are restricted to operations – including conditional operations – that accept a minimum number of constants or memory variables, called *registers*, and assign the result to a register again, e.g., $r_0 := r_1 + 1$.

Analogous to tree-based GP, the name "linear" refers to the structure of the (imperative) program representation. It does not stand for functional genetic programs that are restricted to a linear list of nodes only. Moreover, it does not mean that the method itself is linear, i.e., may solve linearly separable problems only. On the contrary, genetic programs normally represent highly non-linear solutions in this meaning.

The use of linear bit sequences in GP again goes back to Cramer and his JB language [26]. A more general linear approach was introduced by Banzhaf [9]. Nordin's idea of subjecting machine code to evolution was the first GP approach [64] that is directly operating with an imperative representation. It was subsequently expanded and developed into the AIMGP (*Automatic Induction of Machine code by Genetic Programming*) approach [68, 11]. In AIMGP individuals are manipulated as binary machine code in memory and are executed directly without passing an interpreter during the fitness calculation. This results in a significant speedup compared to interpreting systems. Besides the invention of machine code GP, Nordin's thesis [68] focuses on aspects that are relevant to machine code GP and on the application of this linear GP approach to different problem domains. Only some work [65, 67] is dedicated to more general phenomena of linear GP.

This thesis concentrates on fundamental characteristics of the linear program representation and shows differences to the tree representation. Based on such features advanced LGP techniques are developed. All analyses performed and methods presented are supposed to be independent of a special type of imperative programming language or machine language. Nonetheless, a transfer ability of results to machine code GP is preserved as far as possible. Moreover, the methods presented here are not meant to be specific to a certain application area, but may be applied to a wide range of problems, including approximations and classifications in particular.

Basically, there are two major differences that distinguish a linear program from a tree program:

(1) Linear genetic programs feature a graph-based data flow that results from a multiple usage of register contents. That is, on the functional level the evolved imperative structure represents special directed graphs. In traditional GP, instead, the data flow is determined by the tree structure of programs.

The higher variability of program graphs allows the result of subprograms (subgraphs) to be reused multiple times during calculations. On the one hand, this allows linear solutions to be more compact in size than tree solutions and to express more complex calculations with less instructions. On the other hand, the step size of variations may be under better control in a program structure with higher degree of freedom than that of a tree. How much evolution may take advantage of these features strongly depends on the design of appropriate variation operators.

(2) Special noneffective code coexists with effective code in linear genetic programs that results from the imperative program structure – not from program execution – and can be detected efficiently and completely. Such structurally noneffective code manipulate registers not having an impact on the program output at the current position and is, thus, not connected to the data flow generated by the effective code. In a tree program, by definition, all program components are connected to the root. As a result, the existence of noneffective code necessarily depends on program semantics.

Noneffective code in genetic programs is also referred to as *introns*. In general, it specifies instructions without any influence on the program behavior. Noneffective code is argued to be beneficial during evolution for two major reasons. First, it may act as a protection such that it reduces the effect of variations on the effective code. In linear programs introns may be created easily at each position with (almost) the same probability. Second, noneffective code allows (more) variations to remain neutral in terms of a fitness change.

According to the above notions, we distinguish between an *absolute program* and an *effective program* in linear GP. While the first includes all instructions, the latter contains the (structurally) effective instructions only. The (*effective*) *length* of a program is measured in the number of (effective) instructions it holds. Each program *position* or *line* is supposed to hold exactly one instruction. Even if the absolute length of a program has reached the maximum complexity bound it can still vary in size of its effective code. The effective length is especially important because it reflects the number of executed instructions in our approach and, thus, the execution time.

A more detailed introduction to linear GP can be found in Chapter 2. For a detailed description of tree-based GP we refer to Chapter 7 here.

1.4 Motivation and Overview

Up to now, the traditional tree representation of programs is still dominating research in the area of GP, even if many different GP approaches and program representations have appeared in the last years. A general motivation for investigating different representations in evolutionary computation is that for each representation form, as well as for different learning methods in general, *certain* problem domains may exist that are more successfully solved than others. This holds true even if the *No Free Lunch* (NFL) theorem [100] states that there is no search algorithm better, on average, than any other search algorithm over the set $F = \{f : \mathcal{S} \rightarrow W\}$ of *all* functions (problems) f for a finite search space S. In particular, the NFL theorem includes evolutionary algorithms.

A special concern for this thesis is to convince the reader that there are some serious advantages of a linear representation of programs compared to a tree representation. As noted above, linear GP is not only the evolution of imperative programs, but may be reduced to the evolution of special program graphs.

An observation that can be made is that linear GP is often used in applications or for representation-independent GP techniques by researchers, but it is considered less if the basic understanding of GP or the analysis of representation-specific aspects are concerned. One goal of this thesis is to fill this gap at least partly. First, an exhaustive analysis of the linear program representation is performed in terms of its imperative and functional structure. The analysis of program structure at runtime serves as a tool for better understanding the functionality of linear GP. Second, general GP phenomena, such as intron code, neutral variations, and code growth, are investigated for the linear variant.

Another focal point is the enhancement of linear GP on the methodical level. In doing so, the general objective targets are to produce more precise and more efficient prediction models, as this is true for other machine learning approaches. In particular, information about program structure is exploited for various techniques, including, e.g., the acceleration of processing time and the design of efficient genetic operators. Minimization of variation steps on the symbolic level will turn out to be one key criterion for obtaining more successful solutions.

The thesis is organized in 10 chapters that are briefly summarized in the following paragraphs. In the present Chapter 1 a general and more formal introduction to genetic programming has been given, after a short review of the history of evolutionary algorithms.

Chapter 2 describes the linear GP approach in more detail. This includes general concepts as well as the specific LGP variant used in this thesis.

In Chapter 3 efficient algorithms are presented for analysing linear genetic programs in terms of their special properties. This comprises the identification of different types of noneffective code as well as the extraction of information about the underlying functional structure.

Chapter 4 compares the standard LGP approach with an efficient variant of neural networks based on several classification problems from medicine. Additionally, a significant acceleration of processing time is documented for linear GP by eliminating noneffective code from the genetic programs before the fitness evaluation.

The theoretical results of Chapter 3 have inspired the development of more efficient genetic operators for the linear representation in Chapter 5. These lead to a better quality of solutions, in the first place, and to a lower complexity, in the second place. Important points of interest in this context are the variation step size, the amount of noneffective code that emerges in programs, and the proportion of neutral variations.

Moreover, the influence of several variation parameters is analysed in Chapter 5. Chapter 6 reports on how more general control parameters influence the performance and the complexity of solutions, including those in particular that are specifically related to the linear representation, e.g., the number of registers.

In Chapter 7 different variants of linear GP are compared with tree-based GP on both benchmark problems and real-world classification problems from bioinformatics. Linear GP will turn out to be superior, especially when applying more sophisticated operators from Chapter 5.

In Chapter 8 we define structural and semantic distance metrics for linear genetic programs to explicitly control diversity and to reduce the variation step size more precisely on the effective program code.

Chapter 9 deals with the phenomenon of code growth in genetic programming. Different theories about code growth are verified for linear GP. Special importance is attached to neutral variations that are identified as being a main cause of code growth as well as an important motor of evolutionary progress.

One possibility to scale the complexity and the dimension of programs is to evolve multiple independent program modules as one individual. Chapter 10 applies the evolution of such program teams to prediction tasks and compares different methods for combining the multiple team outputs.

Chapter 2

Basic Concepts of Linear Genetic Programming

Contents

In this chapter the reader will be introduced to *linear genetic programming* (LGP) in further detail. This is done on the basis of the specific linear GP variant that is investigated in this thesis. In particular, the evolved programming language, the representation type, and the specific evolutionary algorithm are defined, which form the kernel of the described LGP system. In doing so, basic concepts of linear GP are discussed, including different forms of program execution.

As already indicated in the introduction, linear GP operates with imperative programs. This thesis focuses on the imperative representation in general. That is, all discussions and experiments are conducted independently of a special type of programming language or processor architecture. Even though genetic programs are interpreted and partly noted in the high-level language C the applied programming concepts exist principally in or may be translated into most modern imperative programming languages, including machine languages.

2.1 Representation of Programs

The imperative programming concept, in contrast to the functional programming paradigm, is closely related to the underlying machine language. All modern CPUs are based, in principle, on the von Neumann architecture, where a computing machine is composed of a set of registers and basic instructions that operate and manipulate their contents. A program of such a register machine, accordingly, denotes a sequence of instructions whose order has to be respected during execution.

```
void gp(r)
  double r[8];
{
   ...
   r[0] = r[5] + 71;
// r[7] = r[0] - 59;
   if (r[1] > 0)
   if (r[5] > 2)
     r[4] = r[2] * r[1];
// r[2] = r[5] + r[4];
   r[6] = r[4] * 13;
   r[1] = r[3] / 2;
// if (r[0] > r[1])
//   r[3] = r[5] * r[5];
   r[7] = r[6] - 2;
// r[5] = r[7] + 15;
   if (r[1] <= r[6])
     r[0] = sin(r[7]);
}
```

Example 2.1: LGP program in C notation. Commented instructions have no effect on program output stored in register `r[0]` (see 3.2.1).

Basically, an *imperative instruction* includes an operation on *operand* (or *source*) *registers* and an assignment of the result to a *destination register*. Instruction formats exist for zero[1], one, two or three registers. Most of modern machine languages are based on 2-register or 3-register instructions, however. 3-register instructions operate on two arbitrary registers

[1] 0-register instructions operate on a stack.

(or constants) and assign the result to a third register, e.g., $r_i := r_j + r_k$. In 2-register instructions, instead, either the implemented operator requires only one operand, e.g., $r_i := sin(r_j)$, or the destination register acts as a second operand, e.g., $r_i := r_i + r_j$. Due to a higher degree of freedom a program that contains 3-register instructions may be more compact in size than a program that is built from 2-register instructions only. For that reason and for a higher flexibility we will regard instructions with a free choice of operands only.

In general, we allow at most one operation per instruction with a minimum number of operand – usually one or two. Note that a higher number of operators or operands in instructions would not necessarily increase the expressiveness or the variability of programs. Such instructions would assign the result of a more-or-less complex expression to a register. Moreover, such a two-dimensional program structure would make genetic operations definitely more complicated.

In our LGP system a genetic program is interpreted as a variable-length sequence of simple C instructions. To apply program solutions directly in a problem domain (without using a special interpreter) their internal representation is translated into C code. An excerpt of a linear genetic program, as exported by the system, is given by Example 2.1. In the following, the term "genetic program" always refers to the internal LGP representation that we will discuss in more detail now.

2.1.1 Coding of Instructions

In our implementation all registers hold floating-point values. Internally, constants are stored in registers that are write-protected, i.e., may not become destination registers. As a consequence, the set of possible constants stays fixed. Constants are addressed by indices in the internal program representation just like variable registers and operators (see below). Constant registers are only initialized once at the beginning of a run with values from a user-defined range. One advantage over encoding constants explicitly in the program instructions is that memory space is saved, especially as far as real-valued or larger integer constants are concerned. A continuous variability of constants by the genetic operators is further not absolutely needed and should be sufficiently counterbalanced by interpolation in the genetic programs. Furthermore, a free manipulation of real-valued constants inside programs could result in program solutions that may be exported only imprecisely. Note that floating-point values can only be printed to a certain accuracy. If a program uses many constants, rounding errors may be reinforced during execution on which the overall program behavior may depend.

Each of the maximum four instruction components, including one instruction identifier and three register indices, may be encoded into one byte of memory only. Then the maximum number of variable registers *and* constant registers is restricted to 256, which is, however, absolutely sufficient for most problem definitions. For instance, instruction $r_i := r_j + r_k$ reduces to a vector of indices $< id(+), i, j, k >$. Actually, an instruction is held in a single 32-bit integer value. Such a coding of instructions is similar to a representation as machine code [64] but may be chosen independently of the type of processor in our interpreting system. In particular, the described coding allows an instruction component to be accessed efficiently by casting the integer value (instruction) previously into an array of 4 bytes. A program is represented by an array of integers. This compact representation is not only memory-efficient but allows an efficient manipulation of programs as well as an efficient interpretation (see Section 2.2).

In the following we will refer to a *register* only as a variable register. A constant register is identified with its constant value.

In linear GP a user-defined number of variable registers, the *register set*, is provided to the genetic programs. Besides a minimum required number of *input registers* which hold the program inputs before execution, additional registers can be provided in order to facilitate calculations. Normally these so called *calculation registers* are initialized with a constant value (1 here) each time before a program is executed on the fitness cases. Only for special problem applications, like time series predictions, where an order is defined on the fitness cases it may be advantageous to give this up. If calculation registers are only initialized once before the fitness evaluation they allow an exchange of information between successive executions of the same program for different fitness cases.

A sufficient number of registers is important for the performance of linear GP, especially if the input dimension and the number of input registers, respectively, are low. In general, the number of registers determines the number of program paths (in the functional representation) that can be calculated in parallel. If it is not sufficient there are too many conflicts by overwriting of register information within a program. One or more input registers or calculation registers may be defined as *output registers*. The standard output register is (input) register r_0. The imperative program structure also facilitates the use of multiple program outputs. Instead, functional expressions like trees calculate one output only, by definition (see also Section 7.1).

2.1.2 Instruction Set

The *instruction set* defines the particular programming language that is evolved. In our LGP system this is based on two basic *instruction types* – including operations[2] and conditional branches. Table 2.1 lists the general notation of all instructions that have been used in experiments of this thesis.

Instruction type	General notation	Input range
Arithmetic operations	$r_i := r_j + r_k$	$r_i, r_j, r_k \in \mathbb{R}$
	$r_i := r_j - r_k$	
	$r_i := r_j \times r_k$	
	$r_i := r_j \ / \ r_k$	
Exponential functions	$r_i := {r_j}^{(r_k)}$	$r_i, r_j, r_k \in \mathbb{R}$
	$r_i := e^{r_j}$	
	$r_i := ln(r_j)$	
	$r_i := {r_j}^2$	
	$r_i := \sqrt{r_j}$	
Trigonomic functions	$r_i := sin(r_j)$	$r_i, r_j, r_k \in \mathbb{R}$
	$r_i := cos(r_j)$	
Boolean operations	$r_i := r_j \wedge r_k$	$r_i, r_j, r_k \in \mathbb{B}$
	$r_i := r_j \vee r_k$	
	$r_i := \neg \ r_j$	
Conditional branches	$if \ (r_j > r_k)$	$r_j, r_k \in \mathbb{R}$
	$if \ (r_j \leq r_k)$	
	$if \ (r_j)$	$r_j \in \mathbb{B}$

Table 2.1: LGP instruction types.

[2]Functions will be identified with operators in the following.

Two-operand instructions may either include two indexed variables (registers) r_i as operands or either one operand is a constant (but not both). One-operand instructions only use register operands. In doing so, assignments of constant values, e.g., $r_0 := 1 + 2$ or $r_0 := sin(1)$, are avoided explicitly (see also Section 6.3). That is, we allow not more than one constant per instruction. Then the percentage of instructions holding a constant equals the proportion of constants p_{const} in programs. This is also the probability for which a constant operand is selected during both the initialization of programs and mutations. The influence of this parameter will be analysed in Section 6.3. In most other experiments documented in this thesis $p_{const} = 0.5$ is used.

In genetic programming it must be guaranteed that only valid programs are created. The genetic operators – including recombination and mutation – have to maintain the *syntactical correctness* of newly created programs. In linear GP, for instance, crossover points may not be selected inside an instruction and mutations may not exchange an instruction operator for a register. To assure *semantic correctness* partially defined operators and functions may be protected by returning a high constant value for all undefined inputs, e.g., $c_{undef} := 10^6$. Table 2.2 summarizes all instructions from Table 2.1 that have to be protected from certain input ranges and gives the respective definitions. High results of operations act as a punishment for programs that use these otherwise undefined inputs. If low constant values would be returned, i.e., $c_{undef} := 1$, protected instructions may be exploited more easily by evolution for the creation of semantic introns (see Section 3.2.2). For instance, all instructions preceding effective instruction $r_i := r_j/0$ are semantic introns which only influence the content of register r_j.

Instruction	Protected definition			
$r_i := r_j \; / \; r_k$	$if\ (r_k \neq 0)$	$r_i := r_j \; / \; r_k$	$else$	$r_i := r_j + c_{undef}$
$r_i := {r_j}^{r_k}$	$if\ (\lvert r_k \rvert \leq 10)$	$r_i := \lvert r_j \rvert^{r_k}$	$else$	$r_i := r_j + r_k + c_{undef}$
$r_i := e^{r_j}$	$if\ (\lvert r_j \rvert \leq 32)$	$r_i := e^{r_j}$	$else$	$r_i := r_j + c_{undef}$
$r_i := ln(r_j)$	$if\ (r_j \neq 0)$	$r_i := ln(\lvert r_j \rvert)$	$else$	$r_i := r_j + c_{undef}$
$r_i := \sqrt{r_j}$	$r_i := \sqrt{\lvert r_j \rvert}$			

Table 2.2: Definitions of protected instructions.

To minimize the input range that is assigned to a semantically rather senseless function value, undefined negative inputs are mapped to defined absolute inputs in Table 2.2. This may make it easier for evolution to integrate protected instructions into a robust program semantics. It is also possible not to protect instructions at all but simply punish programs (with the worst fitness) that calculate an infinite or non-numeric (NaN) output value for a fitness case.

On the one hand, the ability of genetic programming to find a solution strongly depends on the expressiveness of the instruction set. On the other hand, the dimension of the search space, i.e., all possible programs that can be built from these instructions, increases exponentially with the number of instructions and registers. A *complete* instruction set contains all elements that are necessary to build the optimum solution at least in principle – provided that the number of variables registers and the range of constants are sufficient. If we take into account that the initial population usually represents a small fraction of the complete search space only, the probability to find the optimum solution or a good approximation decreases significantly with too many useless types of such basic program elements. Finally, the probability for which a certain instruction is selected as well as its frequency in the population influence solution finding. To control the selection probabilities of instruction types more specifically, the instruction set may contain *multiple* *instances* of an instruction.

In this thesis we do not regard program functions that induce *side-effects* to the problem environment, but return a single value only in a strict mathematical sense. Side-effects may be used for solving control problems, for instance. A linear program may represent a list of commands (plan) that directs a robot agent in an environment. The fitness information is then derived from the agents interactions with its environment, i.e., by reinforcement learning. In such a case, the genetic programs do not represent mathematical functions.

2.1.3 Branching Concepts

Conditional branches are an important and powerful concept in genetic programming. In general, programming concepts like branches or loops allow the control flow to be altered, that is given by the structure of the representation. The control flow in linear genetic programs is linear while the data flow is organized as a directed graph (see Section 3.3). When using conditional branches the control flow (and the data flow) may be different for different input situations, i.e., may depend on program semantics.

Usually classification problems are solved more successfully or even exclusively if branches are provided. Branches may increase the (effective) complexity of solutions by promoting a specialization of solutions and by forming semantic introns (see Chapter 3). Both may lead to less robust and less generalizing solutions.

If the condition of a branch instruction, as defined in Table 2.1, is false only *one* instruction is skipped (see also discussion in Section 3.3.2). Sequences of branches are interpreted as *nested branches* in our system (as in C). That is, the next non-branch instruction, i.e., operation, in the program is executed only if all conditions are true and is skipped otherwise. In general, we refer to such a combination of conditional branch(es) and operation as a *conditional operation*:

```
if (<cond1>)
if (<cond2>)
<oper>;
```

Nested branches allow more complex conditions to be evolved and are equivalent to connecting the single branch conditions by a logical AND. A disjunction (OR connection) of branch conditions, instead, may be represented by a sequence of conditional instructions whose operations are identical:

```
if (<cond1>)
<oper>;
if (<cond2>)
<oper>;
```

Alternatively, successive conditions may be interpreted as being connected either by AND or by OR. This can be achieved in the following way: A Boolean operator (AND or OR) is encoded into each branch identifier. This requires the information of a binary flag only, which determines how the condition of a branch instruction is connected to a potentially preceeding one in program. The status of these flags may be varied during operator mutations. Only the transformation of the (internal) representation into a C program becomes slightly more complicated because each sequence of branches has to be substituted by a single branch with an equivalent condition of higher order.

2.1.4 Advanced Branching Concepts

A more general branching concept is to allow conditional forward jumps over a variable number of more than one instruction. The number of skipped instructions may either be unlimited, i.e., limited by the length of program only, or may be selected randomly from a certain range. In the latter case the actual length of a jump may be determined by a parameter that is encoded in each branch instruction (using the identifier section or the unused section of the destination register). It is also possible to do without this additional overhead by using constant block sizes, instead. Because not all instructions of a skipped code block are usually effective, evolution may control the semantic effect of a jump by the number of noneffective instructions within a jump blocks.

A transformation of such branches from the internal program representation into working C code requires constructions like

```
if (<cond>) goto <label X>;
<...>
<label X>;
```

where *unique X* labels have to be inserted at the end of each jump block.

One possibility to avoid branching into blocks of other branches allows jumps not to be longer than the position of the *next* branch in program. In this way, the number of skipped instructions does not have to be administrated within the branches and is limited more implicitly. Translation into C is achieved then simply by setting `{...}` brackets around the jump block.

Another interesting variant is to allow jumps to *any* succeeding branch instruction in program only. This can be realized by using an additional pointer with each branch instruction to an arbitrary successor branch (*absolute jump*). *Relative jumps* to the kth *next* branch in program with $1 \leq k \leq k_{max}$ are also possible, even if such connections are separated more easily if a new branch instruction is inserted or deleted. A pointer to a branch that does not exist anymore may be automatically replaced by a valid pointer after variation. The last branch in programs may always point to the end of program, by default ($k := 0$). Hence, the control flow in a linear genetic program may be interpreted as a *directed acyclic branching graph* (see Figure 2.1). The nodes of such a *control flow graph* represent subsequences of (non-branch) instructions.

Kantschik and Banzhaf [45] propose a more general concept of a branching graph for the imperative representation. Each node contains an instruction block that ends with a single *if-else*-branch. These branches point to two alternative decision blocks which represent two independent successor nodes. Thus, instructions may not only be skipped within an otherwise linear control flow but real parallel subprograms may exist in programs. This form of representation is called a *linear graph* since it defines a graph-based control flow on linear genetic programs. Recall that the term *linear* genetic program derives from the linear flow of control that is given by the linear arrangement of instructions. In Section 3.3 we will see that the data flow is graph-based already in simple linear genetic programs.

In general, a complex non-linear control flow requires either more sophisticated variation operators or repair mechanisms after variation. For the branching graphs a special crossover operator may be constrained so that only complete nodes or subgraphs of nodes are exchanged between programs with a certain probability. That is, crossover points fall upon branch instructions only. Unrestricted linear crossover (see Section 2.3.4) may be applied then between graph nodes (instruction blocks) only.

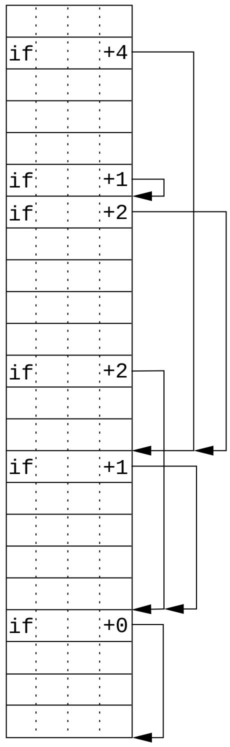

Figure 2.1: Branching graph: Each branch instruction points to an arbitrary succeeding branch.

The final branching concept whose capability is discussed here for linear GP uses an additional `endif` instruction in the instruction set. Nested

```
if (<cond>)
<...>
endif
```

constructions are interpreted such that an `endif` belongs to an `if` counterpart if *no* branch or *only closed* branching blocks lie in between. An instruction that cannot be assigned in this way may either be deleted from the internal representation or contribute to the noneffective code. One advantage of such a concept is that it allows an (almost) unconstrained and complex nesting of branches while jumps into other branching blocks cannot occur. A transformation into C code is achieved simply by setting `{...}` brackets around valid branching blocks instead of `endif` and by not transforming invalid branch instructions. In a similar way `if-else-endif` constructions may be realized, too.

2.1.5 Iteration Concepts

Iteration of code parts by loops rather plays a less important role in genetic programming. Most GP applications that require loops deal with control problems where, in general, the combination of primitive actions of an agent is an object of evolution. There is no (relevant) flow of data in such programs necessary. Instead, each action performs side-effects to a problem environment and fitness is derived from a reinforcement signal. For the problem classes on which this work concentrates, classification and approximation of labeled data, iterations are of minor importance. Nevertheless, a reuse of code by iterations may result in more compact program solutions.

In functional programming the concept of loops is unknown, in principle. The implicit iteration concept in functional programs denotes *recursions* which are rather hard to control in (tree-based) genetic programming. Otherwise, simply iterated evaluations of a subtree can have an effect only if functions produce side-effects. In linear GP assignments represent an implicit side-effect on memory locations as part of the imperative representation. Nevertheless, the iteration of an instruction segment may only be effective if it includes at least one effective instruction *and* if at least one register acts as both destination register and source register in the same or a combination of (effective) instructions, e.g., $r_0 := r_0 + 1$.

In the following, possible iteration concepts for linear GP will be presented. In principle, these comprise *conditional loops* and loops with a *limited* number of iterations.

One form of iteration in linear programs are conditional backward jumps which correspond to a `while` loop in C. The problem with this concept is that it forms infinite loops easily by conditions that are always fulfilled. In general, it is not possible to detect all infinite loops in (genetic) programs. This is due to the theoretical *halting problem* that states we cannot decide whether a program will stop or not [28]. One possible solution is to terminate a genetic program after a maximum limit of executed instructions has been exceeded. But then the result of the program depends on the execution time.

A more recommended loop concept limits the number of iterations specifically for each loop. This requires an additional control flow parameter which may either be constant or be varied within the loop instructions. Such a construction is usually expressed by a `for` loop in C. Because only overlapping of loops, rather than nesting, has to be avoided an appropriate choice to limit the size of loop blocks may be the coevolution of `endfor` instructions in programs. Analogous to the interpretation of branches in Section 2.1.4, a `for` and a succeeding `endfor` instruction define a loop block if *no or only closed* loops lie in between. All other loop instructions are not interpreted.

2.1.6 Modularization Concepts

For certain problems modularization may be advantageous in GP. On the one hand, by using subroutines repeatedly within programs, solutions may become smaller in size. That is, the same maximum program space can be used more efficiently for more powerful solutions. On the other hand, a problem may be decomposed into simpler subproblems that may be solved more efficiently in local submodules. A combination of subsolutions may result in a simpler and better overall solution then.

The most popular modularization concept in genetic programming are so called *automatically defined functions* (ADFs) [52]. Basically, a genetic program is split up into a main program and a certain number of subprograms (ADFs). The main program calculates the program result by using the coevolved subprograms via function calls. Therefore, the ADFs are treated as part of the main instruction set. Each module type may be composed of different sets of program components. It is furthermore possible to define a usage graph that defines which ADF type may call which other ADF type. Usually recursions are avoided by not allowing cycles then. The crossover operator has to be constrained in such a way that only modules of the same type are recombined between two individuals.

ADFs denote an *explicit* modularization concept since the submodules are encapsulated from the main program and may only be used locally in the same individual. Each module is represented by a separate tree expression [52] or a separate sequence of instructions [68].

To assure encapsulation of modules in linear programs disjoint sets of registers have to be used. Otherwise, unwanted state transitions between modules may occur.

ADFs denote subsolutions that are combined by being used in a main program. In this thesis another explicit form of modularization, the evolution of program *teams*, is investigated (see Chapter 10). A team comprises a fixed number of programs that are coevolved as one GP individual. In principle, all member programs of a teams are supposed to solve the same problem by receiving the same input data. These members act as modules of an overall solution such that the member outputs are combined in a predefined way. A better performance may result here from a collective making of decision and specialization of more-or-less independent program modules.

A more *implicit* modularization concept that prepares code for reuse is an automated module *acquisition* [5]. Here certain substructures of a program are identified as modules. Such modules are chosen more-or-less randomly from better individuals by a compression operator and are replaced by respective module calls. The new modules are outhoused into a global library where they may be referenced by any individual of the population. In functional representations a replacement of subexpressions (subtrees) is relatively simple. In linear GP, instead, a subsequence of instructions is always bound to a certain register usage within an imperative program context. If such a module is supposed to be extracted it had to be replaced by a function call that manipulates the same *global* register set as the respective main program.

Complex module dependences may hardly emerge during evolution if modularization is not really needed for better problem solutions. In general, if a programming concept is redundant, the resulting larger search space may influence solution finding rather negatively. Moreover, the efficiency of a programming concept or a program representation in GP always depends on the variation operators, too. Thus, even if the expressiveness or flexibility of a programming concept is high in principle, it may be more difficult for evolution to take advantage of it.

2.2 Execution of Programs

The higher the processing speed of a learning method is the more complex or time-dependent applications may be handled. The most time-critical steps in evolutionary algorithms are the fitness evaluation of individuals and/or the calculation of a new search point (individual) by the variation operators. In genetic programming computation costs are dominated by the fitness evaluation which requires multiple executions of a program, at least one for each fitness case. Executing a genetic program means that the internal program representation is interpreted in a definite way while following the semantics of the programming language that is evolved.

For instance, interpretation in TGP systems works by traversing the tree structure of programs in postorder or preorder. While doing so, operators are applied to operand values that result recursively from executing all subtrees of the operator node first.

In a special variant of linear GP, called AIMGP (*Automatic Induction of Machine code by Genetic Programming*) [64, 11], individuals are represented and manipulated as binary machine code. Because programs can be executed directly without passing an interpreter, machine code GP results in a significant speedup compared to interpreting GP systems. Due to their dependence on specific processor architectures, however, machine systems are restricted in portability. Moreover, machine code system may be restricted in functionality, e.g., in the number of existing hardware registers.

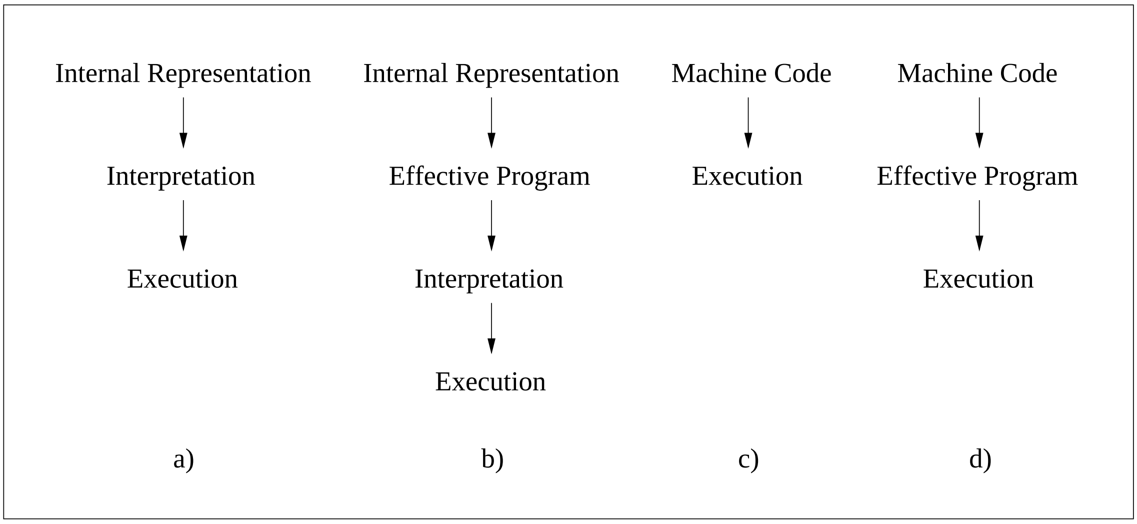

Figure 2.2: Different forms of program execution: (a) Interpretation of programs in GP. (b) Elimination of noneffective code in LGP. (c) Direct execution of machine code in AIMGP. (d) Combination of b) and c).

Another method to accelerate the execution (interpretation) of linear genetic programs is applied in this thesis. The special type of noneffective code, that results from the imperative program structure, may be detected efficiently in linear runtime (see algorithm in Section 3.2.1). In our LGP system noneffective code is removed from a program before its fitness is calculated, i.e., before the resulting effective program is executed over multiple fitness cases. By doing so, the evaluation time of programs may be reduced significantly, especially if a larger number of fitness cases is processed. In the example program from Section 2.1 all commented instructions are noneffective if program outputs are stored in register `r[0]`.

Since AIMGP is a special variant of linear GP, both acceleration techniques may be combined in such a way that a machine code representation is preprocessed by a routine extracting the effective parts. This results in four different ways of processing in genetic programming that are illustrated in Figure 2.2.

An elimination of introns can be relevant only, of course, if a significant amount of this code is created by the variation operators. In particular, this is true for linear crossover (see Section 2.3.4). An additional acceleration of runtime in linear GP results from the fact that the fitness of an individual has to be recalculated only if the (structurally) effective code has undergone change. Instead of the evaluation *time*, this method may reduce the *number* of evaluations (and program executions) that are performed during a generation (see Section 5.2).

2.2.1 Runtime Comparison

The following experiment gives an impression of the differences in processing speed that may occur with the four ways of program execution in linear GP (see Figure 2.2). To guarantee a fair comparison between machine code GP and interpreting GP, an interpreting routine has been added to an AIMGP system. This routine *interprets* the machine code programs in C so that they produce exactly the same results as without interpretation. Both interpreting and non-interpreting runs of the system are accelerated by a

Parameter	Setting
Problem type	polynomial regression
Number of fitness cases	200
Number of runs	10
Number of generations	200
Population size	1000
Maximum program length	256
Maximum initial length	25
Crossover probability	90%
Mutation probability	10%
Operator set	$\{+,-,\times\}$
Number of registers	6
Set of constants	$\{0,..,99\}$

Table 2.3: Parameter settings

second routine that removes the noneffective code. In Table 2.3 general settings of system parameters are given for a polynomial regression task.

Table 2.4 compares the average *absolute* runtime for the four different configurations with respect to interpretation and intron elimination. If interpretation is not applied, programs are executed directly as machine code. 10 runs have been performed for each configuration while using the same set of 10 different random seeds. In doing so, the runs behave exactly the same for all configurations apart from their processing speed. Note that the average length of programs in the population exceeds 200 instructions in about generation 100. The intron rate converges to about 80%, on average.

Runtime (sec.)	No Interpretation (I_0)	Interpretation (I_1)
No Intron Elimination (E_0)	500	6250
Intron Elimination (E_1)	250	1375

Table 2.4: Absolute runtime in seconds (rounded) averaged over 10 independent runs (on a SPARC Station 10)

The resulting *relative* speed factors are listed in Table 2.5. In contrast to the absolute runtime these values are independent of the number of processed fitness cases. If both the direct execution of machine code *and* the elimination of noneffective code are applied in combination runs become about 25 times faster for the considered problem and system configuration. Note that the influence of intron elimination on the interpreting runs (factor 4.5) is more than two times bigger than on the non-interpreting runs (factor 2). This reduces the advantage of machine code GP over interpreting GP from a factor of 12.5 to a factor of 5.5. Standard machine code GP without intron elimination occurs to be less than 3 times faster than linear GP including this extension.

Apparently, the performance gain by the intron elimination strongly depends on the proportion of (structurally) noneffective instructions in programs. In contrast to the size of effective code, this is less influenced by the problem definition than by the variation operators and the system configuration (see Chapters 5 and 6).

$E_0I_0 : E_0I_1$	1 : 12.5
$E_1I_0 : E_1I_1$	1 : 5.5
$E_0I_0 : E_1I_0$	1 : 2
$E_0I_1 : E_1I_1$	1 : 4.5
$E_0I_0 : E_1I_1$	1 : 2.75
$E_1I_0 : E_0I_1$	1 : 25

Table 2.5: Relative runtime for the four configurations of Table 2.4.

2.2.2 Translation

From an application point of view the best (generalizing) program solution denotes the only relevant result of a GP run. Of course, the internal representation (coding) of this program could be exported as it is. Then, however, an interpreter is required to guarantee that the program will behave in the application environment as it did in the GP system. To avoid this programs are exported as equivalent C functions in our LGP system (see Example 2.1 and Figure 2.3). It has already been pointed out in Section 2.1.2 how single programming concepts are transformed into C. In general, by translating internal programs into an existing (imperative) programming language, solutions may be integrated directly into an application context (software) without additional overhead.

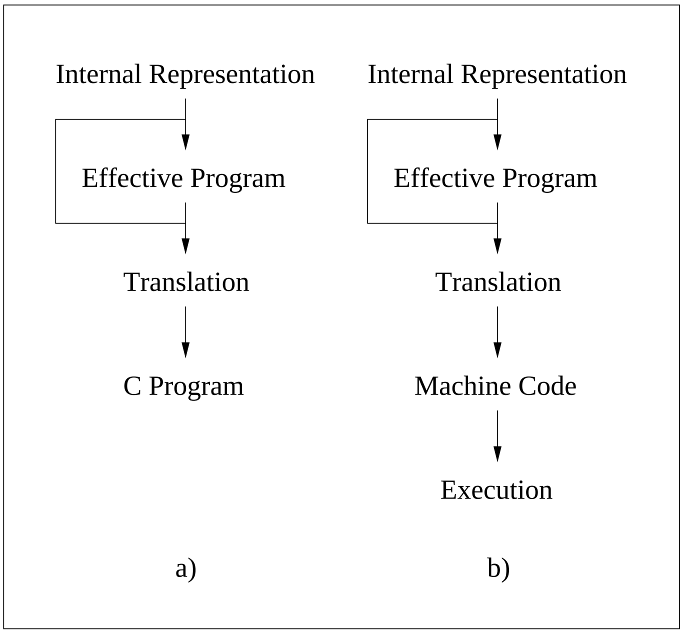

Figure 2.3: (a) Translation into C program. (b) Translation into machine code.

Another benefit of such a translation is that it allows less restrictions to be imposed on the internal representation. Instead, the representation may be chosen (almost) freely, e.g., in favor of a better evolvability and a better variability in GP. Since usually only a few individuals are exported during a run even complex transformations may not be time-critical.

The same advantage – higher flexibility – together with a higher processing speed may motivate a translation from the evolved LGP language into a binary machine language (*compilation*) only before the fitness of a program is evaluated (see Figure 2.3). Note that the direct manipulation of machine programs in AIMGP systems is less important for runtime. Instead, the speed advantage mostly results from the direct execution of machine code. At least code translations from an imperative language should be possible efficiently, especially if the noneffective code is removed before.

2.3 Evolutionary Algorithm

Algorithm 2.1 describes the evolutionary algorithm that builds the kernel of our LGP system. In a *steady-state* EA, like this, there are no fixed generations defined, in contrast to a *generational* EA. For the latter variant, the current generation is identified with a population of parent programs which offsprings migrate to a *separate* population pool. After the offspring pool is fully populated it replaces the parent population and the next generation begins. In the steady-state model there is no such centralized control of generations. Instead, offsprings replace existing individuals in the *same* population. It is a common practice to define *generations* in steady-state EAs artificially as regular intervals of fitness evaluations. Only newly created individuals have to be evaluated if the fitness is saved with each individual in the population. Usually one generation is completed if the number of new individuals equals the population size.

ALGORITHM 2.1 (*LGP algorithm*)

1. *Initialize* a population of random programs (see Section 2.3.1).
2. Randomly *select* $2 \times n$ individuals from the population without replacement.
3. Perform two *fitness tournaments* of size n (see Section 2.3.2).
4. Make *temporary copies* of the two tournament winners.
5. *Modify* the two winners by one or more variation operators for certain probabilities (see Section 2.3.4).
6. *Evaluate* the fitness of the two offsprings.
7. If the currently best-fit individual is replaced by one of the offsprings *validate* the new best program using unknown data.
8. *Reproduce* the two tournament winners within the population for a certain probability or under a certain condition by replacing the two tournament losers with the temporary copies of the winners (see Section 2.3.3).
9. Repeat steps 2. to 8. until the maximum number of generations is reached.
10. *Test* the program with minimum validation error again.
11. Both the best program during training and the best program during validation define the output of the algorithm.

The *fitness* of an individual program is computed by an *error function* on a set of input-output examples $(\vec{i_k}, o_k)$. These so called *fitness cases* define the problem that is desired to be solved or to be approximated by the genetic programs. A popular error function for approximation problems is the *sum of squared errors* (SSE), i.e., the squared difference between the predicted output $gp(\vec{i_k})$ and the desired output o_k for all n training examples

A squared error function punishes larger errors more than smaller errors. Equation 2.1 defines the *mean squared error* (MSE). For classification tasks the *classification error* (CE) calculates the number of wrongly classified examples. Function *class* in Equation 2.2 hides the classification method that maps the continuous program outputs to discrete class identifiers. While a better fitness means a smaller error the best fitness is 0.

$$\mathrm{MSE}(gp) = \frac{1}{n}\sum_{k=1}^{n}(gp(\vec{i_k}) - o_k)^2 \tag{2.1}$$

$$\mathrm{CE}(gp) = \sum_{\substack{class(gp(\vec{i_k}))\neq o_k \\ k=1,..,n}} 1 \tag{2.2}$$

The generalization ability of individual solutions is checked *during* training by calculating the *validation error* of the currently best-fit program with the same error function. This unknown *validation data set* is sampled differently from the training data, but from the same data space. Finally, among all the best individuals emerging over a run the one with minimum validation error (point of best generalization) is tested on an unknown *test data set*, again once *after* training is over. Note that a validation of the best solutions follows a fitness gradient. Validating all individuals during a GP run is not reasonable, since we are not interested in solutions that perform well on the validation data but have a comparatively bad fitness. Moreover, this would produce higher computational costs that cannot be neglected.

If an individual is selected for variation or if it is ruled out by others depends on relative fitness comparisons during selection. In order not to loose information a copy of the individual with minimum validation error has to be saved outside of the population. The individual with minimum training error (best individual) cannot be overwritten as long as the training data is fixed during evolution.

Training data may be resampled every mth generation or even each time before an individual is evaluated. On the one hand, resampling introduces noise into the fitness function (*dynamic fitness*). This is argued to improve the generalization performance compared to keeping the training examples constant over a run because it reduces *overtraining*, i.e., an overspecialization of solutions to the training data. On the other hand, resampling may be beneficial if the data base is large that constitutes the problem to be solved. A relatively small subset size may be used for training while all data points will be exposed to the genetic programs over time. As a result, not only the fitness evaluation of programs is accelerated but the evolutionary process may converge faster, too. This technique is called *stochastic sampling* [11].

2.3.1 Initialization

In normal case, the initial population of genetic programs is built up complete randomly. In linear GP an upper bound for the initial program length has to be defined. The lower bound may be identically equal to the absolute minimum length of a program which is one instruction. When a program is created its length is chosen randomly from that predefined range for a uniform probability.

On the one hand, it is not recommended to initialize programs too long, as will be demonstrated in Section 6.6. This may reduce their variability significantly in the course of the evolutionary process. Besides, the smaller the initial programs are, on average, the more thorough the exploration of the search space may turn out at the beginning of a run.

On the other hand, the average initial length of programs should not be too small, because a sufficient diversity of the initial genetic material is necessary, especially in smaller populations or if crossover dominates variation.

2.3.2 Selection

Algorithm 2.1 applies *tournament selection*. With this selection method individuals are selected randomly from the population to participate in a tournament where they compete for the best fitness. Normally selection happens without replacement, i.e., all individuals of a tournament must be different. The *tournament size* n_{ts} determines the selection pressure that is imposed on the population individuals. If a tournament is held between *two* individuals (and if there is only one tournament used for selecting the winner) this corresponds to the minimum selection pressure. A lower pressure is possible with this selection scheme only by performing $m > 1$ tournaments and choosing the *worst* among m winners.

In the LGP algorithm always *two* tournaments happen in parallel to provide two parent individuals for crossover. For comparison reason, this is practiced also if mutations are applied exclusively (see Chapter 5). Before the tournament winners undergo variation, a copy of each winner is taken that replaces the (worst) loser of a tournament. Such a reproduction within the population constitutes a steady-state EA.

Tournament selection, together with a steady-state evolutionary algorithm, is well suited for parallelization by using more-or-less isolated subpopulations of individuals, called *demes* (see also Section 4.3.2). Tournaments may be performed independently of each other and do not require global information about the population, like a global fitness ranking (*ranking selection*) or the average fitness (*fitness proportional selection*) [17]. Local selection schemes are argued to better preserve the diversity than global selection schemes. Moreover, individuals may take part in a tournament several times or not at all during one steady-state generation. This allows evolution to progress with different speeds in different regions of the population.

2.3.3 Reproduction

A full reproduction of winners guarantees that better solutions always survive in a steady-state population. However, during every replacement of individuals a certain amount of genetic material gets lost. When using tournament selection this situation can be influenced over the *reproduction rate* p_{rr}. By using $p_{rr} < 1$ the EA may *forget* better solutions to a certain degree. Both reproduction rate and selection pressure (tournament size) have a direct influence on the convergence speed of the evolutionary algorithm as well as on the loss of (structural and semantic) diversity.

Another possible alternative to the standard reproduction rate ($p_{rr} = 1$) is to allow $p_{rr} > 1$. That is, an individual will be reproduced *more than once* within the population, on average. A sufficiently large tournament size is required here to provide enough worse individuals (losers) which may be replaced by the multiple copies of the tournament winner, i.e., $n_{ts} > \lceil p_{rr} \rceil$. As a result, both the convergence speed and the loss of diversity may be accelerated accordingly. Obviously, too many replications of individuals lead to an unwanted premature stagnation of the evolutionary process. Note that more reproductions are performed than new individuals are created.

Instead of or in addition to an explicit reproduction probability, more implicit conditions can be checked under which reproduction shall take place (see Section 9.5).

2.3.4 Variation

Genetic operators change the contents and the size of genetic programs in the population. Figure 2.4 illustrates the *two-point linear crossover* as it is used in linear GP for recombining two genetic programs [11]. A segment of random position and arbitrary length is selected in each of the two parents and exchanged. In our implementation (see also Section 5.7.1) crossover exchanges equally sized segments if one of the two children would exceed the maximum length, otherwise.

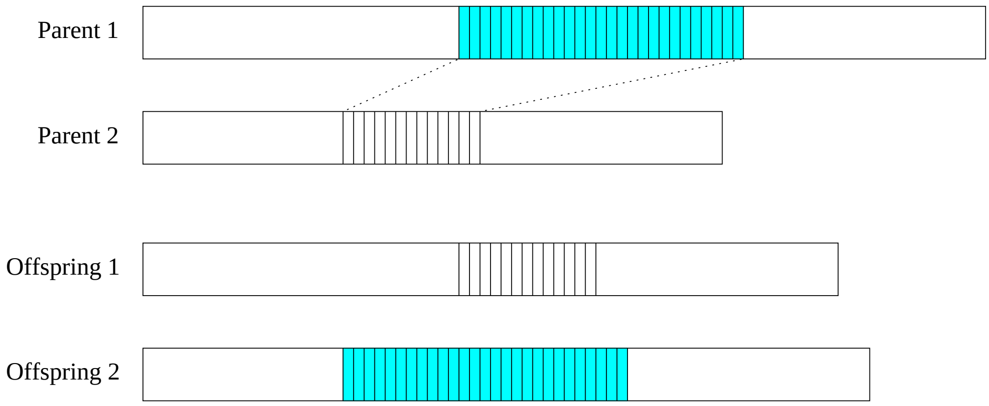

Figure 2.4: Crossover in linear GP. Continuous sequences of instructions are selected and exchanged between parents.

Crossover is the standard *macro operation* that is applied to vary (the length of) linear genetic programs on the level of instructions, i.e., instructions are the smallest units to be changed. *Inside* instructions *micro mutations* randomly replace either the instruction identifier, a register or a constant (if existent) by equivalents from predefined sets or valid ranges. In Chapter 5 we will introduce more advanced genetic operators for the linear program representation.

It may be guaranteed for each variation that it modifies the program structure. Therefore, identical exchanges of code have to be avoided explicitly. These are, however, not very likely when using crossover, especially if the length of exchanged segment is unrestricted.

In general, there are three different ways in which variation operators may be selected and applied to a certain individual program before its fitness is (re)calculated:

- Only *one* variation is performed per individual.
- *One* variation operator is applied *several* times.
- *More than one* variation operator is applied.

One advantage of using only one genetic operation per individual is a lower total variation strength. This allows artificial evolution to progress more specifically and in smaller steps. By applying several genetic operations concurrently, on the other hand, computation time is saved such that less evaluations are necessary. For example, micro mutations are often applied together with a macro operation.

Note that in all three cases, there is only one offspring created per parent individual, i.e., only one offspring gets into the population and is evaluated. Analogous to a multiple reproduction of parents as discussed in Section 2.3.3, one may derive more than one offspring from a parent, too. Both is, however, not practiced by Algorithm 2.1.

Chapter 3

Characteristics of the Linear Representation

Contents

Originally linear genetic programming has been introduced for the benefit that the genetic programs can be executed (as binary machine code) without passing a time-consuming interpretation step first (see Section 2.2). Apart from this speed advantage, we investigate other, more general characteristics of the linear representation in this chapter. As already mentioned in the introduction, one basic difference compared to a tree representation is that unused code parts occur and remain within linear genetic programs that are independent from program semantics. Another difference is the data flow in a linear genetic program that describes a directed graph, i.e., is not restricted to a tree structure.

3.1 Effective Code and Noneffective Code

Introns in nature are subsequences of DNA strings holding information that is not expressed in the phenotype of an organism or, more precisely, that is not translated into a protein sequence. The existence of introns in eucaryotic genomes may be explained in different ways: (1) Since the information for one gene is often located on different *exons*, i.e., gene parts that are expressed, introns may help to reduce the number of destructive recombinations between chromosomes by simply reducing the probability that the recombination points will fall within an exon region [97]. In this way, complete protein segments encoded by specific exons are more frequently mixed than interrupted during evolution. (2) Perhaps even more important for understanding the evolution of higher organisms is the realization that new code can be developed "silently" without exposing each intermediate variation step to fitness selection.

In genetic programs there may be code parts that are either essential or redundant for the program solution. Redundant code fragments are called *introns* [1] like its natural counterpart. Actually, introns in GP may play a similar role as introns in nature. First, introns reduce the destructive influence of variations on the effective part of programs. In doing so, they may protect the information holding code from being separated and destroyed. Second, the existence of noneffective code allows code variations to be neutral in terms of a fitness change. This retains genetic manipulations from direct evolutionary pressure. In linear GP we distinguish effective instructions from noneffective instructions.

Definition 3.1 (*effective/noneffective instruction*) An instruction of a linear genetic program is *effective* at its position iff it influences the output(s) of the program for at least one possible input situation. A *noneffective* or *intron* instruction, respectively, is without any influence on the calculation of the output(s) for all possible inputs.

One noneffective instruction is regarded as the smallest unit. A noneffective instruction may be removed from a program without affecting its semantics – either independently or only in combination with other noneffective instructions. In analogy to biology an *intron* in LGP may be defined as any instruction or combination of instructions where this is possible. A second, weaker intron definition that is distinguished in this thesis postulates the program behaviour to be unchanged only for the fitness cases [67].

Definition 3.2 (*noneffective instruction*) An instruction of a linear genetic program is *noneffective* iff it does not influence the program output(s) for the fitness cases.

The condition in Definition 3.2 does not necessarily hold for unknown data inputs. If the generalization performance of best individuals is checked during training and some of

[1] Even if intron code is redundant for a certain problem solution, this is not necessarily true for the evolutionary process of solution finding.

these introns would be removed before the validation error is calculated, the behavior of the program may not be the same anymore.

Definition 3.3 (*effective/noneffective register*) A register is *effective* for a certain program position iff its manipulation can effect the behavior, i.e., an output, of the program. Otherwise, the register is *noneffective* at that position.

Effective instructions following Definition 3.1 necessarily manipulate effective registers (see Definition 3.3). But an operation can still be noneffective even if its result is assigned to an effective register.

In this thesis we favor single conditional instructions as introduced in Section 2.1.3. Then a branch instruction is effective only if it directly precedes an effective instruction. Otherwise it is noneffective. That is, a conditional instruction is effective as a whole if this is true for its operation.

3.2 Structural Introns and Semantic Introns

The above considerations suggest an additional classification of introns in linear GP. This is based on a special type of noneffective code that results from the imperative structure of programs – not from program semantics. Hence, two types of noneffective instructions may be discerned: structural introns and semantic introns.

Definition 3.4 (*structural intron*) *Structural* or *data flow introns* denote single noneffective instructions that emerge in a linear program from manipulating noneffective registers.

Actually, the term *structural intron* refers to the functional structure of linear genetic programs that constitutes a directed graph, as will be demonstrated in Section 3.3. Structural introns belong to a part of the graph that is not connected to the (effective) root node which calculates the program output. That is, these instructions do not contribute to the *effective data flow*. Structural introns do not exist in tree-based GP, because in a tree structure, by definition, all program components are connected to the root. Thus, introns in tree programs result from the program semantics. In linear GP semantic introns may be defined as follows:

Definition 3.5 (*semantic intron*) A *semantic* or *operational intron* is a noneffective instruction or a noneffective combination of instructions even though it manipulates effective register(s).

That is, a semantic intron is necessarily (structurally) effective by this definition. Otherwise it would be a structural intron. The state of effective registers manipulated by a semantic intron is the same before and after the intron has been executed – if we assume that operations do not induce side-effects. For instance, instruction $r_0 := r_0 \times 1$ is a semantic intron if register r_0 is effective. While all structural introns are noneffective after Definition 3.1 and Definition 3.2, semantic introns may be noneffective after Definition 3.2 only. But note that not all semantic introns depend necessarily on the fitness cases. More examples of semantic introns will be given in Section 3.2.2.

According to Definitions 3.4 and 3.5 we distinguish *structurally effective* code from *semantically effective* code. While the first type may still contain semantic introns the latter code is supposed to be intron-free. However, even if all intron instructions can be removed from a program, it has not necessarily a minimum size (see Section 3.2.4).

Alternatively, when regarding only Definition 3.1 structural introns may also be designated as *neutral noneffective* code and semantic introns as *neutral effective* code, respectively. Such a naming conforms to the distinction of *neutral noneffective* variations and *neutral effective* variations, as will be defined in Section 5.1.1. It has to be noted, however, that neutral code does not only result from neutral variations (see Chapter 9) which produces confusing names. The different intron definitions will become more clear in the following sections.

Whether a branch is a structural intron or a semantic intron depends again on the status of the operation that directly follows. Semantic introns include branch instructions, too, whose condition is always true, at least for all fitness cases. In this case, all other branches are skipped that follow directly in a sequence (nested branch, see Section 2.1.3). Such *non-executed* instructions represent special semantic introns. An operation is not executed if the condition of a directly preceding (nested) branch is always false.

3.2.1 Detecting and Removing Structural Introns

In biology introns are removed from the *messenger*-RNA, a copy of the DNA, that actually participates in gene expression, i.e., protein biosynthesis [97]. A biological reason for the removal of introns might be that genes are more efficiently translated during protein biosynthesis in this way. Without being in conflict with ancient information held in introns, this might have an advantage, presumably through decoupling of DNA size from direct evolutionary pressure.

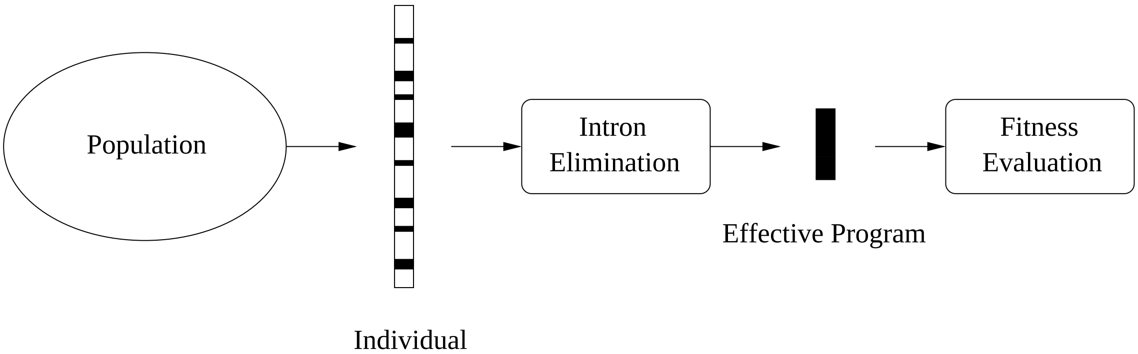

Figure 3.1: Intron elimination in LGP. Only effective code (black) is executed.

The imperative program structure in linear GP permits (structurally) noneffective instructions to be identified efficiently. This in turn allows the corresponding effective instructions to be extracted from a program during runtime and to be copied to a temporary program buffer once before the fitness of the program is calculated (see Figure 3.1). By only executing this effective program when testing each fitness case, evaluation can be accelerated significantly. Thereby, the representation of individuals in the population remains unchanged while the computation time for the noneffective code is saved. No potential genetic material gets lost and the intron code may fulfill its functions during the evolutionary process (see above). In analogy to the elimination of introns in nature, the linear genetic code is interpreted more efficiently. Because of this analogy the term "intron" might be more justified here than in tree-based GP where introns are necessarily semantic and, thus, may be detected much harder (see below).

Algorithm 3.1 detects all structural introns in a linear genetic program that does not apply loops (backward jumps) or jumps over more than one instruction (see Chapter 2.1). More generally, such an *elimination of dead code* represents a form of code optimization that is applied, for instance, during compilation [1]. The algorithm includes a simple *dependence analysis* that identifies all instructions on which the final program output depends directly or indirectly. All effective, i.e., depending, instructions are marked in programs by using one bit of the instruction coding (see Section 2.1.1) as an *effectiveness flag*. Copying all marked instructions at the end forms the *effective program*. In the example program from Section 2.1 all instructions marked with an `//` are structural introns provided that the program output is stored in register `r[0]` at the end of execution.

Algorithm 3.1 (*detection of structural introns*)

1. Let set R_{eff} always contain all registers that are effective at the current program position. $R_{eff} := \{ r \mid r \text{ is output register } \}$.
   Start at the last program instruction and move backwards.
2. Mark the next preceding operation in program with destination register $r_{dest} \in R_{eff}$. If such an instruction is not found then $\rightarrow$ 5.
3. If the operation directly follows a branch or a sequence of branches then mark these instructions too. Otherwise remove r_{dest} from R_{eff}.
4. Insert each source (operand) register r_{op} of newly marked instructions in R_{eff} if not already contained. $\rightarrow$ 2.
5. Stop. All unmarked instructions are introns.

The algorithm needs linear calculation time $O(n)$ with n is the program length. Actually, detecting and removing the noneffective code from a program only requires about the same time as calculating one fitness case. The more fitness cases are processed by the resulting effective program the more this computational overhead will pay off. A good estimate of the overall acceleration in runtime is the factor

$$\alpha_{acc} = \frac{1}{1 - p_{intron}} \tag{3.1}$$

with p_{intron} the average percentage of intron code in a genetic program and $1 - p_{intron}$ the respective percentage of effective code.

By omitting the execution of noneffective instructions during program interpretation a large amount of computation time can be saved. A removal of structural introns may be relevant only, of course, if a sufficient proportion of this noneffective code occurs with the applied variation operators (see Chapter 5). System parameters like the maximum program length influence this proportion because effective length may grow even after absolute length has reached the maximum. Moreover, the creation of structural introns is facilitated if a higher number of registers is provided. If only one register is available, this type of code cannot occur at all. We will demonstrate in Section 6.1 that both too less or too many registers may influence the prediction performance negatively. In general, the intron rate depends less on the problem since (the size of) this code is not directly affected by the fitness selection.

3.2.2 Avoiding Semantic Introns

As noted above, structural introns may be identified completely by Algorithm 3.1, but the resulting effective code may still include semantic introns. In general, a detection of semantic introns is much more difficult and may only be incomplete (see Section 3.2.4). As an inherent part of the program structure, the structurally noneffective code is not directly depending on the applied set of instructions. Moreover, this type of noneffective code may be implemented easily by linear genetic programming even in great quantities. Structural introns take away a lot of pressure from the genetic programs to develop semantic introns as a reduction of the variation step size on the (semantically) effective code (see Chapters 5.9.1 and 9). Moreover, since structural introns may be detected and removed efficiently they allow (effective) solutions to be more compact in size and, thus, save computation time.

The proportion of semantic introns may be further reduced by controlling the formation of this code more explicitly. Even if these introns cannot be avoided completely in genetic programming some rules can be observed that avoid at least simple possibilities to create semantic introns without restricting the freedom of variation or the expressiveness of the function set significantly. The harder it becomes for the system to develop noneffective code that depends on program semantics, the more this code should be ruled out by structural introns.

The potential of linear GP to develop semantic introns strongly depends on the provided set of instruction operators and the set of constants. To restrict the rate of semantic introns and, thus, to keep the (structurally) effective size of programs small, both sets may be chosen with a minimum tendency for creating semantic introns. Below different *types of semantic introns* are given by example, that are possible with instruction set $\{+, -, \times, /, x^y, if >, if \leq\}$ (see Table 2.1), together with some rules how each type may be avoided at least partly. The intron classes are not meant to be necessarily disjoint. Some examples may be borderline cases, i.e., fit in more than one class. All semantic introns denote noneffective code for *all* possible input situations (following intron Definition 3.1). We do not regard instructions as introns here that are noneffective for certain input ranges or the fitness cases only (see Definition 3.2). In the following register r_0 is supposed to be effective (otherwise introns would be structural).

(1a) $r_0 := r_0 + 0$

(1b) $r_0 := r_0 \times 1$

(1c) $r_0 := {r_0}^1$

(1d) $r_2 := r_0 + r_0$
$r_1 := r_2 - r_0$
$r_0 := r_1 + 0$

Semantic introns of type (1) become less likely if constants 0 and 1 are not explicitly provided to act as neutral elements in operations. It is especially cheap and effective to do without constant 0, since it is not really useful for calculation but has a high potential for creating semantic introns:

(2a) $r_0 := r_i \times 0$

(2b) $r_0 := {r_i}^0$

(2c) $r_1 := r_0 - r_0$
$r_0 := r_i \times r_1$

Not these example instructions, but at least one preceding instruction in program that influences the content of no other effective register than r_i is a semantic introns of type (2). This intron type can include many noneffective instructions. Note that even if value 0 is excluded from the set of constants it may still be calculated and assigned to a variable register, independent from the register contents (see context examples (1d) and (2c)). However, the more complex such intron constructs become the more context-dependent they are and the more likely they will be destroyed during variation.

(3a) $r_0 := r_i - r_i$

(3b) $r_0 := r_i \ / \ r_i$

(3c) $r_1 := r_i + c$
$r_0 := r_1 - r_i$

Introns of type (3) result from registers like r_0 whose contents becomes constant by calculation, i.e., does no longer dependent on other register variables. If r_0 is the only effective register at a program position, all preceding instructions will be introns. Otherwise, all preceding instructions are introns that manipulate register r_i exclusively. The reader may recall that instructions with only constant operands are not possible (see Section 2.1.1). One operand is always variable. To make the creation of type (3) introns more difficult direct subtraction and division of identical registers might be forbidden explicitly.

(4) $r_1 := r_0 + 1$
$r_0 := r_1 - 1$

The above example represents an intron of type (4). It includes all *combinations* of instructions that may be symbolically simplified without requiring any (semantically equivalent) replacement through other instructions (see Section 3.2.4). The same is true for type (1) introns that comprise a single instruction only. Such introns are difficult to avoid in general, especially if more larger redundant calculations are involved. It may be questioned, however, if complex context-dependent introns occur frequently and survive during program evolution.

Register r_1 has to be noneffective at the position of intron example (4) in a program. Otherwise, these instructions might not be removed without changing the (effective) program. In general, all registers that are manipulated in semantic introns must be either (structurally) noneffective or their original contents before the intron is restored after the last instruction of the intron has been executed.

(5a) $r_0 := r_i \ / \ 0$

(5b) $r_1 := r_0 - r_0$
$r_0 := r_i \ / \ r_1$

Typically, the undefined range of a protected operator is exploited for the induction of type (5) introns. This variant can be avoided by punishment as described in Section 2.1.2.

(6a) $if\ (r_i > r_i)$
$r_0 := r_j + c$

(6b) $r_2 := r_i + r_i$
$r_1 := r_2 - r_i$
$if\ (r_1 > r_i)$
$r_0 := r_j + c$

(6c) $r_0 := r_i + 2$
$r_1 := r_0 - r_i$
$if\ (r_1 \leq 1)$
$r_0 := r_j + r_k$

(6d) $if\ (r_i > 2)$
$if\ (r_i \leq 1)$
$r_0 := r_j + r_k$

Type (6) is a special case of semantic intron. The operation is not executed at all because the branching condition cannot be met. As a result, all preceding instructions become noneffective, too, whose effectiveness depends only on the skipped instruction. Example (6a) cannot occur if equal registers are not allowed to be compared. More context-dependent conditions (6b) are not affected by such a restriction, but are created less likely. Other conditions (6c) that are unsatisfiable for all possible register values emerge from comparisons of constant values. Note again that direct comparisons of two constants are avoided explicitly. A conjunction of contradicting conditions (6d) emerges less likely if only one comparison is provided to the system. By doing so, the expressiveness of the programming language is not restricted significantly. Alternatively, sequences of branches might be explicitly forbidden.

(7a) $if\ (r_i \leq r_i)$
$r_0 := r_j + c$

(7b) $r_1 := r_i + 2$
$r_0 := r_1 - r_i$
$if\ (r_0 > 1)$
$r_0 := r_j + r_k$

Type (7) represents the opposite case to type (6). That is, a conditional operation is always executed because the condition is always true. Here the branch instruction itself is an intron as well as all preceding instructions that are effective only in the false case.

(8) $if\ (r_1 > 1)$
$if\ (r_1 > 1)$
$r_0 := r_j + r_k$

Finally, redundant branch instructions that may occur in nested branches constitute introns of type (8).

3.2.3 Detecting Semantic Introns

The specific measures proposed in the previous section reduce the probability that semantically noneffective code occurs in linear genetic programs. It is generally not necessary and not affordable to apply expensive algorithms that detect and remove semantic introns explicitly during runtime. Usually the evolutionary process is already accelerated significantly by eliminating the larger number of structural introns (see Algorithm 3.1).

Nevertheless, a removal of semantic introns makes sense for a better understanding of a certain program solution and to gain information about the application domain, in this way. Another motivation to further reduce the (structurally) effective size *after* evolution may be a higher efficiency in time-critical application domains.

Algorithms that detect certain types of (structural or semantic) noneffective code as specified by Definition 3.1 are better deterministic. Probabilistic algorithms that require the execution of a program necessarily depend on a more-or-less representative set of input-output examples. Such algorithms may identify instructions whose intron status depends on certain input situations (see Definition 3.2). Since normally not all possible inputs can be verified for a problem, such intron instructions may become effective when being confronted with unknown data.

The following probabilistic algorithm (similar to the one documented in [11]) detects semantic introns. All structural introns, instead, are detected as a side-effect even if much more inefficiently than by Algorithm 3.1. Hence, computation time may be saved if the program is already free from structural introns.

Algorithm 3.2 (*elimination of semantic introns*)

1. Calculate the fitness $\mathcal{F}_{ref}$ of the program on a set of m data examples (fitness cases) as a reference value.
   Start at the first program instruction at position $i := 1$.
2. Delete the instruction at the current program position i.
3. Evaluate the program again.
4. If its fitness $\mathcal{F} = \mathcal{F}_{ref}$ then the deleted instruction is an intron.
   Otherwise, reinsert the instruction at position i.
5. Move to the next instruction at position $i := i + 1$.
6. Stop, if the end of program has been reached. Otherwise $\rightarrow$ 2.

Algorithm 3.2 needs calculation time $O(m \cdot n^2)$ because of n fitness evaluations, $m+1$ program executions per fitness evaluation, and n (effective) program instructions at maximum. This is too inefficient for removing introns during runtime. The higher computational costs would hardly be paid by the savings obtained during the fitness evaluation.

Unfortunately, Algorithm 3.2 will not recognize semantic introns that are more complex than one instruction (see Section 3.2.2). One possibility to find all semantic introns in a linear genetic program for a certain set of fitness cases (following Definition 3.2) is to repeat the algorithm for all k-party combinations of arbitrary program instructions with $k = 1, 2, .., n$.

3.2.4 Symbolic Simplification

Introns have been defined in Section 3.1 as single instructions or combinations of instructions that may be removed without replacement and without affecting program semantics. But even if a linear genetic program is completely free from semantic and structural introns, the size of the remaining (semantically) effective code is not necessarily minimum. The following example (9) is not an intron, but may be referred to as a *mathematically equivalent extension*. It represents all formulations of a subprogram that are more complicated than necessary. Such combinations of instructions cannot be removed, but may be replaced by less complex, semantically equivalent code.

(9) $r_0 := r_0 + 1$
$r_0 := r_0 + 1$
$\Leftrightarrow$
$r_0 := r_0 + 2$

A (structurally effective) program can be transformed into a functional tree expression by a successive replacement of variables (see Section 3.3.4) provided that program operators do not induce side-effects. During such a transformation process the expression can be simplified successively by applying rules of *symbolic calculation*. In doing so, semantic intron instructions by Definition 3.1 are removed deterministically. The probabilistic Algorithm 3.2, instead, removes noneffective code by Definition 3.2 only and does not resolve mathematically equivalent extension.

In general, detecting absolutely *all* noneffective code and mathematically equivalent extensions is an insolvable problem. Reducing a program to an equivalent of minimum size corresponds to the more general problem whether two programs are equivalent or not. This *program equivalence problem* is in general undecidable because it may be reduced to the undecidable *halting problem* [1, 28]. However, in GP we normally regard finite programs. If no loops or only loops with a finite number of iterations are permitted (see Section 2.1.5), genetic programs will always terminate. Then we may assume that at least theoretically all (semantic) introns can be detected. Unfortunately, already the reduction of an expression to an equivalent expression of minimum size (unique except for isomorphism) is NP-complete [1]. This is true because the NP-complete *satisfiability problem* may be reduced to this *simplification problem*. A general Boolean expression will be unsatisfiable if and only if it simplifies to false.

In the following let the terms *intron* or *noneffective* instruction always denote a structural intron unless stated otherwise. Accordingly, *effective* programs still include semantic introns. As we will see below, the modification of an instruction may change the effectiveness status of other preceding instructions in a linear program – comprising both deactivations and reactivations. Therefore, the terms *active* and *inactive* code will be used as synonyms for effective and noneffective code.

3.3 Graph Interpretation

The imperative representation of a linear program can be transformed into an equivalent functional representation by means of Algorithm 3.3. The directed structure of the resulting graph better reflects functional dependences and data flow in linear genetic programs. The graph is acyclic if loops do not occur in the imperative program. Special cases of programming concepts like loops and branches shall be excluded from the following considerations for simplicity. Instead, we concentrate on the transformation of linear genetic programs as sequences of simple operations into *directed acyclic graphs* (DAGs). It has to be assumed also that program operators/functions do not induce side-effects in the problem environment. Otherwise, the (linear) execution order of instructions may be less flexible than this is required here.

Algorithm 3.3 (*transformation of a linear genetic program into a DAG*)

1. Start with the last instruction in program at position $i := n$ (n = program length). Let set $S := \emptyset$ always contain all variable sinks of the intermediate graphs.

2. If destination register $r_{dest} \notin S$ then create a new start node (a new contiguous graph component) with label r_{dest} and $S := S \cup \{r_{dest}\}$.

3. Go to the (variable) sink node in the graph with label r_{dest}.

4. Assign the operator of instruction i to this node.

5. Repeat steps 6. to 8. for each operand register r_{op} of instruction i:

6. If there is no (variable or constant) sink node with label r_{op} then create a new node with that label.

7. Connect nodes r_{dest} and r_{op} by a directed edge. (r_{dest} becomes inner node and r_{op} becomes sink node.)

8. If not all operations are commutative then label this edge with k if r_{op} is the kth operand.

9. Replace r_{dest} in S by all non-constant operand registers r_{op} of instruction i if not already contained.

10. If $i > 0$ then go to instruction $i := i - 1$ in program and $\rightarrow$ 2.

11. Stop. Delete all register labels from inner nodes.

The number of imperative instructions corresponds exactly to the number of inner nodes in the program graph resulting from Algorithm 3.3. Each inner node represents an operator and has as many outgoing edges as there are operands in the corresponding imperative instruction, i.e., one or two here (see Section 2.1). Thus, each program instruction is interpreted as a small subtree of depth one.

Sink nodes, i.e., nodes without any outgoing edges, are labeled with register identifiers or constants. The number of these *terminals* is restricted by the total number of (different) registers and constants in the terminal set. In a tree representation a terminal may occur multiple times since each node is referenced only once, by definition.

Only sink nodes that represent a (variable) register are replaced regularly by operator nodes in the course of the algorithm. These are the only points at which the graph may grow. Since loops are not considered, the only successors of such sink nodes may become other existing sink nodes or new nodes. At the end of the transformation process these sinks represent the input variables of the program. Note that the data flow in such functional programs runs in the opposite direction in which the edges point.

Sink nodes that represent a constant value are only created once during the transformation process and may be pointed to from every program position. The same is true for constant inputs. Those are held in write-protected registers that may not become destination registers. In doing so, the input information cannot get lost during calculations in the imperative program.

A DAG that results from applying Algorithm 3.3 may be composed of several *contiguous components*. Each of such subgraphs has only one start node from where all its other nodes are reached by at least one (directed) path. *Start* nodes have indegree 0. There may be as many start nodes (contiguous components) in the DAG as there are instructions in the imperative program. The last instruction in program that manipulates an output register corresponds to a start node that initiates an *effective component*. If there is only one output register defined, exactly one graph component is effective. The rest of the graph is *noneffective*, i.e., corresponds to the noneffective instructions (structural introns).

The different contiguous components of a DAG may either be disconnected or may overlap in parts by forming a *weakly contiguous component*. We define that in the latter case all operator nodes are connected (disregarding the direction of edges) but may not necessarily be reached from the same start node (on a directed path). We also let a *non-contiguous* DAG still be weakly contiguous.

Note that noneffective components are not necessarily disconnected from an effective component. Graph edges may point from a noneffective (operator) node to an effective (operator) node, but not the other way around. Thus, noneffective components cannot influence the program output, i.e., the data flow in the effective component which is directed from the sinks to (effective) start node (*effective data flow*). Also note that all components (including disconnected ones) still share the same set of sink nodes in this graph representation.

In the following we assume that the linear program is fully effective in terms of Definition 3.4 and that only one output (register) is defined. Such a program is translated into a DAG that is composed of only a single contiguous component whose start node may also be denoted as the *root* of the DAG.

After each iteration of Algorithm 3.3 all non-constant sink nodes correspond exactly to the effective registers at the current program position. In particular, set S is equal to set R_{eff} in Algorithm 3.1. Because the number of effective registers is limited by the total number of registers, the number of variable sink nodes is limited as well. This number determines the width of the program graph. Since it is usually recommended to use a moderate number of registers, the program graph is supposed to grow in depth. The depth is restricted by the length of the imperative program because each imperative instruction corresponds to exactly one inner node in the graph. For that reasons the graph structure may be referred to as "linear" like the imperative equivalent.

The actual width of a program graph indicates the number of parallel calculation paths in a linear genetic program. It can be approximated by the maximum or the average number of registers that are effective at a program position (see also Section 3.4). Recall that the performance of linear GP strongly depends on a sufficient number of registers. The less registers are available, the more conflicts may occur by overwriting of information during calculations. The more registers are provided, instead, the more local sets of registers may be used for calculating more independent program paths.

It follows from the above discussion that the runtime of Algorithm 3.3 is $O(k \cdot n)$ with n is the number of effective instructions and k is the number of registers. If the total number of (input) registers is small, runtime is approximately linear in n.

b := **c** ∧ 1
c := ¬ **a**
a := c ∨ b
c := b ∧ b
b := c ∨ 1 (x)
a := a ∧ c (x)
c := a ∧ b
b := a ∨ c
a := b ∨ c

Example 3.1: Effective imperative program using Boolean operator set $\{\wedge, \vee, \neg\}$. Output and (used) input registers of the program are bold printed.

The linear program in Example 3.1 corresponds exactly to the DAG in Figure 3.2 after applying Algorithm 3.3. Both the imperative representation and the functional represen-

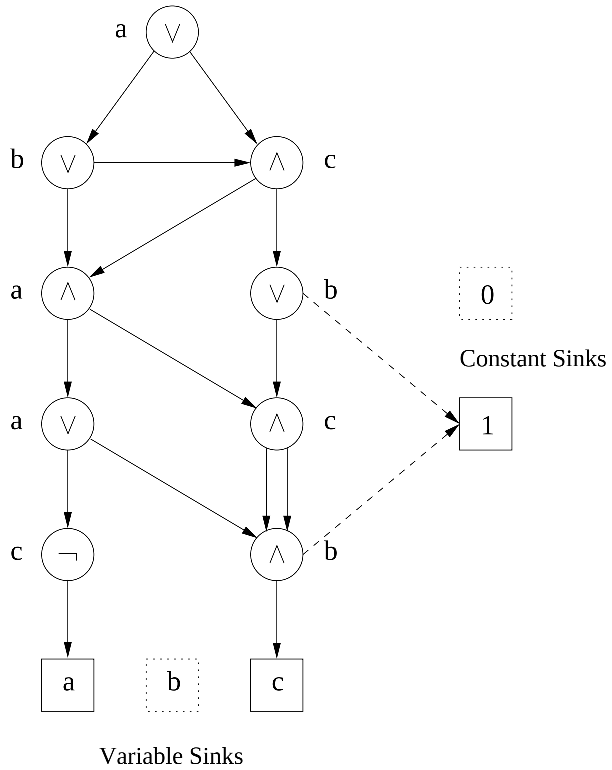

Figure 3.2: Functional equivalent to the effective imperative program in Example 3.1. Operator nodes are labeled with the destination registers of the corresponding instructions (see Algorithm 3.3). Output register **a** marks the start node. (Outgoing edges are not labeled because the order of operands is arbitrary here.)

tation consist of (structurally) effective code here that is free from unused instructions or non-visited graph components, respectively. This is valid if we assume that the output of the imperative program is stored in register **a** at the end of execution. In Example 3.1 only two of the three possible inputs are used. At the beginning of program execution these inputs are held in registers **a** and **c**. *Used* program inputs designate all register operands here that are directly read out before overwritten. In the corresponding graph representation used inputs denote sink nodes (terminals).

3.3.1 Variation Effects

In linear GP already small mutations of the imperative representation, especially the exchange of a register, may have an influence on the functional program structure and the data flow, respectively. Even if the absolute program structure is altered only slightly, the effective program may change drastically. Many instructions preceding the mutated one may be deactivated or reactivated.

Other micro mutations that exchange an operator or a constant can only effect the semantics of a linear program. This is true at least if all operators have the same number of operands. A tree structure does not allow mutations that redirect single edges within a program. At least, this is not possible without loosing the underlying subtree. In program graphs as described above these minimum *structural* or *data flow mutations* are possible due to both their weaker constraints and due to the existence of non-contiguous components.

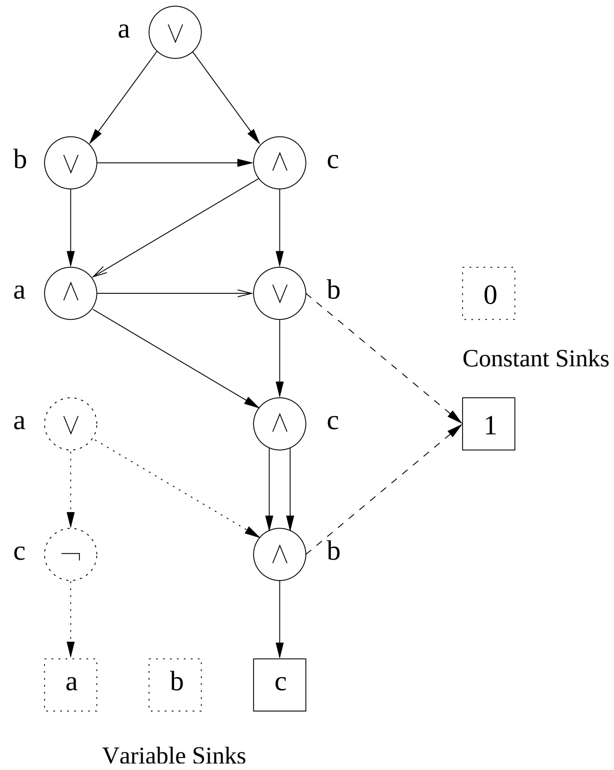

Figure 3.3: Graph interpretation of example program 3.2. Graph is effective except for the dotted component.

Example 3.2 demonstrates the effect of a register mutation on the program from Example 3.1. In particular, the first operand register **a** has been exchanged by register **b** in the sixth instruction from the top. After that two former effective instructions (marked with an (i)) are deactivated, i.e., are identified as (structural) introns now by Algorithm 3.1. Applying Algorithm 3.3 to this program results in the modified graph that is shown in Figure 3.3 and includes a noneffective (and weakly connected) component now. In general, by changing an *operand* register on imperative program level a single edge is redirected in the corresponding graph. The exchange of a *destination* register, on the other hand, may comprise more redirections of edges, instead.

b := c ∧ 1
c := ¬ a (i)
a := c ∨ b (i)
c := b ∧ b
b := c ∨ 1
a := **b** ∧ c
c := a ∧ b
b := a ∨ c
a := b ∨ c

Example 3.2: Linear program from Example 3.1 after register mutation. Operand register **a** has been exchanged by register **b** in the 6th line. Instructions marked with an (i) are structural introns.

3.3.2 Interpretation of Branches

Throughout this thesis we restrict ourselves to the simple branching concept from Section 2.1.3 that considers single conditional operations only. These have a minimum effect on the imperative control flow but may change the data flow in genetic programs significantly. Conditional instructions have a high expressive power because leaving out or executing a single instruction can deactivate much of the preceding effective code or reactivate preceding noneffective instructions, respectively. Actually, an operation that follows a branch may depend on a sequence of preceding instructions that are more-or-less independent from the rest of a program. On the functional level a subgraph is executed in the true case that is (partly) different from the one in the false case. How large this difference may be depends on the total number of registers. As indicated above, the more registers are available the more likely instructions operate on different sets of registers, i.e., the less likely the different data flows intersect.

A single branch instruction is interpreted as an *if-else* node in a functional representation with a maximum four successor nodes: one or two successors for the condition plus one successor each for its true or false outcome. In the true case the conditioned operation is executed and overwrites a certain register contents. In the false case the previous contents of this register remains the current one, i.e., the corresponding calculation is connected to the following data flow.

a := c $\wedge$ 1
b := c $\vee$ 0
if (b)
a := b $\vee$ c

Example 3.3: Conditional branch.

All instructions in Example 3.3 constitute a branching node plus context that is printed in Figure 3.4. We assume that register **a** and, thus, all instructions are effective. If condition **b** = 0 is true in program line 3, the value of register **a** that is calculated in the 1st line influences the following program code. Otherwise, this is the value of **a** in the last line.

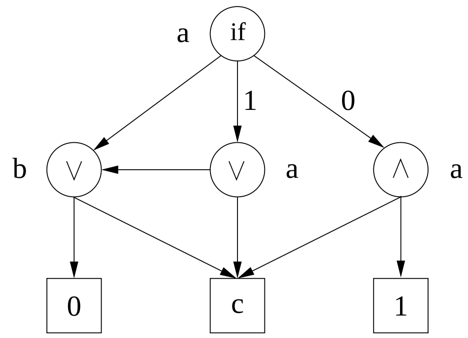

Figure 3.4: Functional equivalent to the conditional branch in Example 3.3. Edge followed in true (false) case is labeled with 1 (0).

Conditional jumps over single instructions in linear GP are at least as powerful in terms of the modification of data flow as branch nodes in tree-based GP. In both approaches only *one* point in data flow is affected. A conditional jump over more than one instruction, by comparison, would be interpreted as multiple branching nodes with identical conditions.

Accordingly, *several* branching points (program paths) are affected simultaneously on the functional level.

But even if such conditional segments are more powerful than conditional instructions, they may suffer from some serious drawbacks in terms of genetic programming. First, changing multiple branching nodes simultaneously may be more difficult to handle by evolution. This is especially true if jumps over branches into other conditional segments are allowed in the imperative programs. Then the control flow in linear programs becomes rather chaotic. Second, larger jumps induce larger variation steps if the branching condition is modified or if the branch instruction is removed. Whole branching blocks may suddenly be executed or not executed for (almost) all possible inputs. This makes both a stepwise improvement of solutions and a reduction of variation step size (as proposed in Chapter 5) more difficult. Third, conditional jumps over more than one instruction have a higher potential for creating semantic introns and, thus, produce larger (structurally) effective programs. As noted in Section 3.2.2 semantic introns may result from branching conditions that cannot be fulfilled. Because of these arguments we may assume that the use of single conditional instructions does not necessarily lead to more restrictive GP solutions than larger conditional segments.

3.3.3 Evaluation Order

In general, calculation in imperative programs results from a sequence of transitions between different states of registers. In a pure functional program there are only values given, but no assignments. Instead, assignments to temporary variables (a stack) are required during the interpretation of programs. In imperative programming these assignments are already included into the program representation.

If a functional genetic program is executed the evaluation order of nodes depends on the way the graph is traversed. This way is not unique because the successor nodes of an inner node may be visited in arbitrary order – if we exclude functions with side-effects again. As in trees the evaluation of nodes in a (contiguous) DAG may be performed in postfix order or prefix order. If the subgraphs of all outgoing edges have been processed, i.e., if all operand values are calculated, the result of a node can be computed. Because subprograms may be used more than once in a graph – in contrast to a tree – the result of evaluation should be saved in each node in order not to evaluate subgraphs twice. The final program result is stored at the root, the only node without incoming edges.

In an imperative genetic program the evaluation order is determined by the linear sequence of instructions. By using advanced programming concepts, like loops or conditional branches, the execution order (*control flow*) of instructions may differ from the linear structural order. The instruction order of a program may be varied in parts without leading to a different program behavior. This is true for both effective and noneffective instructions. For instance, the order of the two effective instructions marked with an (x) may be inverted in Example 3.1, without altering the (effective) data flow or the output of the program. In fact, a functional transformation of the program, if modified like this, will result in exactly the same graph as shown in Figure 3.2. In general, any reordering of instructions is valid here that preserves all dependences in a program, i.e., does not change the execution order (relative position) of depending instructions.

While the imperative structure arranges all instructions in a certain order, such an order is not defined in a functional representation what makes the latter more invariable. As a result, only the transformation of a linear program into a graph is unique (except for isomorphism), but not vice versa.

Another reason why the imperative structure of programs is less unique lies in the fact that internal register identifiers, that are used temporarily during calculations, may be replaced by others without changing the program behavior. During graph interpretation by Algorithm 3.3 those variables label the inner operator nodes only temporarily.

The structural order of operands as well as the number of operands have to be respected in the imperative as well as in the functional representation, at least if instructions depend on it.

3.3.4 Tree Interpretation

An effective linear program can be transformed into a functional expression by a successive (and necessarily multiple) replacement of register variables starting with the last effective instruction. The result of this instruction is necessarily assigned to an output register. If there is more than one program output defined a tree expression is developed correspondingly for each output register.

In order to transform the noneffective imperative code, too, the whole process has to be restarted from the last non-processed instruction in program until all instructions have been processed. Except for the respective last instruction, instructions may be processed more than once. Because each component of the resulting functional program occurs as a separate tree (expression) here, the whole linear genetic program is represented as a *forest*.

These tree programs normally contain many identical subtrees. The deeper a tree node is located the more frequently its corresponding subtree occurs. The size of a tree grows exponentially with the program length n: Let there be only 1 register and only operations with 2 register operands in the imperative program. (Then all instructions are necessarily effective.) The corresponding tree representation is perfectly balanced and contains $2^n - 1$ operator nodes and 2^n (identical) terminal nodes. The corresponding effective graph, by comparison, has only as many nodes as there are effective instructions (n) plus 1 terminal node.

On the one hand, this calculation example demonstrates the high expressive power of linear genetic programs. On the other hand, graph solutions may be more compact in size than tree solutions because subgraphs can be reused several times. The reuse of calculated register contents may also be taken as an argument why ADFs may be less important in linear GP than in tree-based GP [69]. The same may be true in part for the use of iterations in linear GP (see Section 2.1.5).

Only because the constraints of a graph structure are weaker, we may not conclude automatically that linear GP is more powerful than tree-based GP. In general, this does not only depend on the potential variability or expressiveness of a representation but on the design of appropriate genetic operators, too (see Chapter 5).

3.4 Analysis of Program Structure

In this section algorithms are described that extract information about the specific structure of a linear genetic program. All algorithms operate directly on the imperative representation that is a representation for the special program graphs, as demonstrated in the previous section. Three different characteristics are analysed that all refer to the effective part of program.

First, there is the actual *number of effective registers* at an effective or absolute program position. As already mentioned above this information is provided by means of Algorithm

3.1. If set $R_{eff}(i)$ holds all registers that are effective at a position i then $\frac{1}{n+1}\sum_{i=0}^{n} R_{eff}(i)$ denotes the average number of effective registers in a program of n instructions (and $n+1$ intermediate positions). As noted in Section 3.3, this value corresponds approximately to the average width of the (effective) graph equivalent.

In a tree program each node is reached via a unique path from the root, i.e., each node has indegree 1 except for the root (indegree 0). In a graph-structured program, instead, many program paths may lead to the same node, i.e., the indegree of a node is restricted only by the total number of nodes n times the maximum outdegree m of a node. The more narrow a graph develops the more program paths lead through an operator node.

b := c ∧ 1
c := ¬ a
a := c ∨ **b**
c := **b** ∧ **b**
b := c ∨ 1
a := a ∧ c
c := a ∧ **b**
if (**b**)
b := a ∨ c
a := **b** ∨ c

Example 3.4: Linear program from Example 3.1 with branch. All dependences of register **b** are bold printed. The dependence degree is 3 for the 1st and the 5th instruction from the top and 1 for the second last instruction.

Algorithm 3.4 calculates the *degree of effectiveness* in a (structurally) effective program (see Definition 3.6). Each of the $d_{eff}(i)$ operands guarantees that operation i is (structurally) effective. In other words, an operand register guarantees the effectiveness of the next preceding assignment to this register that is not conditional *and* of all conditional assignments to this register that lie in between (see Example 3.4). On the functional level the effectiveness degree corresponds to the number of edges that come into an instruction node, i.e., the *connection degree* or, more precisely, the *indegree* of the node.

Definition 3.6 (*degree of effectiveness/dependence*) The *degree of effectiveness* or *dependence* of an effective operation denotes the number of operand registers in (succeeding) instructions that directly use its result. Let the dependence degree of a branch instruction be identically equal to the dependence degree of its conditioned operation.

The runtime of Algorithm 3.4 is bounded by $O(n^2)$ with n being the effective program length. In worst case no instruction depends on the other. On average, however, runtime can be expected much shorter since usually a register will be used several times (temporarily) as a destination register or operand register, especially if only a few registers are available. In best case each instruction only depends on the instruction that directly follows while computational costs are linear in n. This is true, for instance, if only one program register is used. If Algorithm 3.4 is applied to determine the effectiveness degree of a single instruction only it requires computation time $O(n)$.

Algorithm 3.4 (*degree of effectiveness/dependence*)

1. Assume that all n instructions of a program are effective after Definition 3.4.
   Start at the last instruction in program at position $i := n$ and move backwards.

Let $d_{eff}(i)$ denote the *effectiveness* of an instruction position i.
$d_{eff}(i) := 0$ for $i = 1, .., n$.

2. If instruction i is a branch then $d_{eff}(i) := d_{eff}(i+1)$ and $\rightarrow 7$.
3. $j := i$.
4. If $j < n$ then go to instruction $j := j + 1$. Otherwise $\rightarrow 7$.
5. If destination register $r_{dest}(i)$ of instruction i equals m operand registers $r_{op}(j)$ in instruction j then $d_{eff}(i) := d_{eff}(i) + m$.
6. If neither instruction j nor $j - 1$ are branches and $r_{dest}(i) = r_{dest}(j)$ then $\rightarrow 7$. Otherwise $\rightarrow 4$.
7. If $i > 0$ then go to instruction $i := i - 1$ and $\rightarrow 2$.
8. Stop. The average effectiveness degree of program instructions is defined as $D_{eff} := \frac{1}{n} \sum_{i=1}^{n} d_{eff}(i)$.

Finally, Algorithm 3.5 calculates the average *effective dependence distance* in a program (see Definition 3.7). On the one hand, it gives information about the relative position of depending instructions to each other within an effective imperative program. Since loops are not regarded, an instruction necessarily follows the instructions in program whose result it uses.

DEFINITION 3.7 (*effective dependence distance*) The *effective dependence distance* denotes the relative distance (in effective instructions) of an effective instruction to the first succeeding instruction that depends on it.

On the other hand, this parameter indicates how similar the position of an instruction in an imperative program is to the position of its corresponding node in the functional graph. Two depending instruction nodes are always directly connected in the functional graph. The closer these instructions are in the imperative code, on average, the more similar are the relative positions of instructions and nodes. It follows from Algorithm 3.3 that the last instruction of an effective linear program forms the root of its equivalent directed graph. Theoretically, however, single instructions may be located high up in the effective program while their corresponding node is close to the graph root.

ALGORITHM 3.5 (*effective dependence distance*)

1. Assume that all n instructions of a program are effective after Definition 3.4.
   Start at the first non-branch instruction at a position i.
   Let $\delta_{eff}(i)$ denote the distance between instruction i and the next instructions depending on it.
   $\delta_{eff}(i) := 0$ for $i = 1, .., n$.
2. $j := i$.
3. $\delta_{eff}(i) := \delta_{eff}(i) + 1$.
4. If $j < n$ then go to instruction $j := j + 1$. Otherwise $\rightarrow 6$.
5. If the destination register of instruction i equals an operand register in instruction j then $\rightarrow 6$. Otherwise $\rightarrow 3$.

6. Go to the next succeeding instruction $i := i + k$ ($k \geq 1$) that is not a branch. If this does not exist then $\rightarrow$ 7. Otherwise $\rightarrow$ 2.

7. Stop. The average distance of two depending instructions is $\Delta_{eff} := \frac{1}{n} \sum_{i=1}^{n} \delta_{eff}(i)$.

Algorithm 3.5 resembles Algorithm 3.4 by its basic structure and in runtime.

The effective dependence distance is not only influenced by the instruction order but also by the number and the usage of registers. The minimum distance of two depending instructions is one which is always true if only one register is used. In this case, the functional graph equivalent is reduced to a linear list of operator nodes, each connected by one or two edges. On the one hand, the more registers are provided the more registers may be effective and the wider the functional graph may develop (see above). Both wider graphs as well as longer graphs necessarily require a longer imperative representation. But only for wider graphs the average dependence distance increases because it is less likely that two depending instructions will occur one after another in the imperative program. On the other hand, the more complex the register dependences are, i.e., the higher their dependence degree is, the less variable the order of effective instruction becomes. This may decrease the effective dependence distance. At least if the number of registers is small, we may assume that the position of instructions in the imperative code corresponds approximately to their position in the functional program.

3.5 Graph Evolution

Since the imperative representation may be interpreted as a special graph representation, too, linear GP is reducible to the evolution of program graphs. A question that may arise in this context is whether a direct evolution of a (less constrained) DAG representation may be more advantageous. In the imperative representation the (register) dependence of two instructions is influenced by both their position in the program and the dependences of the instructions that lie in between.

We have seen above that the exchange of a single operand register may reactivate or deactivate other preceding instructions. Former effective (active) instructions become noneffective (inactive) since no other dependence to an effective instruction exists than the one that has been canceled. All such deactivated instructions form a single contiguous graph component of the DAG that is disconnected from the effective component because the only existing connection has been removed.

If variations would happen directly on program graphs this offers a higher degree of freedom in connecting nodes. If single edges may be redirected without restrictions on a functional level, the corresponding changes on the imperative code level may comprise much more complex transformations than exchanging a single register identifier only. This is not only true if cycles are created.

As already noted, the imperative representation defines an (evaluation) order on the effective and the noneffective instructions. This order does not exist in the graph representation where the evaluation order is less constrained and only determined by the connections of nodes.

On the one hand, the imperative order determines and restricts the possible functional connections. A connection to (the destination register of) a preceding instruction is not possible, at least by exchanging just a single register operand. Because registers are used multiple times temporarily in a program, only the next preceding assignment to a certain register may be used in this way. The more registers are provided, however, so much the less this means a relevant restriction of variability. In principle, all transformations are possible on the imperative representation, too, even if larger and more variation steps may be required.

On the other hand, it has to be noted that a higher variability of the representation does not automatically guarantee to find a better solution. Too many degrees of freedom may become disadvantageous. By coevolving an order of instruction nodes in linear GP not only the number of possible connections is restricted but promising connections may be better preserved, too. At least, the probability is increased that functionally disconnected nodes will be reconnected again in the evolutionary process. Also note that a limitation of connections supports the emergence of (structurally) noneffective code, i.e., non-contiguous components.

The most important property, however, is that a linear order of operations implicitly avoids cycles of register dependences by allowing instructions to use only the result of *previous* instructions in program.

If graph structures are evolved without avoiding the formation of cycles, they may not terminate by themselves but execution has to be stopped after a maximum number of visited nodes. During genetic variations it has to be paid attention that all operator nodes receive the correct number of inputs. Depending on whether edges point in data flow direction or usage direction, either the correct number of incoming or outgoing edges has to be checked.

If this is not done and nodes are connected freely the evaluation order of nodes becomes indefinite and a stack (or another state memory) is needed to determine both the exchange of data (data flow) between nodes and the decision which path is visited next (control flow) [96, 18]. That is, the evaluation order has to be explicitly coevolved with these graphs. Side-effects to a state memory may also be used to guarantee that a node is visited only a finite number of times in such control flow graphs.

If an evolved graph structure is supposed to be acyclic without further restricting the freedom of node connections, this has to be verified explicitly after each variation. The detection of all cycles in a graph is, however, computationally expensive. In contrast to that such constraints do not have to be observed during variation in linear GP but result implicitly from the linear sequence of instructions. For the same reason, recombination is much less complicated between linear sequences of instructions than between graphs. It is important to realize that the freedom of variation in DAG evolution is not much higher than in linear GP if cycles are supposed to be avoided. Actually, the freedom of node connections has to be restricted similarly by defining an order on the graph nodes if cycles shall be avoided and, thus, an expensive cycle detection. For instance, Miller [60] evolves acyclic graph programs whose nodes are arranged on a two-dimensional grid. A node in column i is allowed to connect to a node of a *larger* column index $i < j < n$ only that is limited by a maximum distance n.

Finally, a direct evolution of DAGs allows a single contiguous component to be developed exclusively. In Chapter 5 we will introduce variation operators that achieve this for the imperative representation by adding or removing effective instructions only.

3.6 Conclusion

The properties of the special LGP representation as it is applied in this thesis may be summarized as follows:

- On the imperative level a linear genetic program represents a sequence of instructions that comprise single operations or conditional operations with a minimum number of operands. This implies that the control flow is always forward-directed.

- On the functional level a linear genetic program describes a directed acyclic graph (DAG) with a minimum outdegree per operator node. The indegree of nodes is unrestricted in principle. From this it follows that the data flow in linear genetic programs is graph-based.

- Linear GP allows structural noneffective code to coexist in programs that results from manipulating unused registers. In the corresponding graph structure this code may be composed of several disconnected or only weakly connected subgraphs. The effective code forms a connected graph component, instead, if the genetic programs return one output only.

- All operators used in linear genetic programs are mathematical functions without side-effects. That is, a genetic program itself always represents a function.

A linear program defined like this may still be transformed into a tree expresssion. Since each tree is a special DAG, too, this is achieved by copying all subgraphs successively whose start node has more than one incoming edge (starting with the root).

We showed different algorithms in this chapter that extract features from linear genetic programs about their functional or imperative structure. This includes the detection of structural introns which is possible in runtime $O(n)$ with n is the number of instructions. Moreover, an algorithm was presented that transforms a linear program into a DAG. Other more specific features comprise the:

- Number of effective registers

- Degree of dependence (effectiveness)

- Effective dependence distance

The number of effective registers at a certain program position may serve as an approximation for the width of the effective graph component. The width of a graph component is limited by the maximum number of available registers. The effectiveness degree of an instruction corresponds to the indegree of an effective graph node. The distance of an effective instruction to the first succeeding instruction (in the effective program) that depends on it, instead, has no equivalent on the functional level.

Chapter 4

A Comparison with Neural Networks in Medical Data Mining

Contents

The ability of a learning model to generalize, i.e., to predict the outcome of unknown input situations, is an important criterion when comparing the performance of different machine learning methods. This is all the more true for real-world applications in data mining. This Chapter compares the generalization performance of LGP on several medical classification problems with results obtained by neural networks using RPROP learning.

Furthermore, two methods are applied for the acceleration of LGP: (1) The absolute runtime is reduced by using Algorithm 3.1 for the elimination of noneffective code. (2) The effective training time is reduced on a generational basis by means of a deme approach and an elitist migration strategy. Both the time that is necessary for learning a prediction model and the time for its execution become especially important when operating with large datasets as they occur in medical applications.

4.1 Medical Data Mining

Genetic programming and artificial neural networks (NNs) can be seen as alternative techniques for the same tasks, like, e.g., classification and approximation problems. In the analysis of medical data neural networks have become an alternative to classical statistical methods in recent years. Ripley and Ripley [77] have reviewed several NN techniques in medicine including methods for diagnosis and prognosis tasks, especially survival analysis. Most applications of NNs in medicine refer to classification tasks. A comprehensive list of medical applications of neural networks can be found in [14].

In contrast to NNs, GP has not been used very extensively for medical applications to date. Gray *et al.* [33] report from an early application of GP in cancer diagnosis where the results had been found to be better than with a neural network. In [63] a grammar-based GP variant is used for knowledge extraction from medical databases. Rules for the diagnosis have been derived from the program tree that uncover relationships among data attributes. The outcomes of different types of classifiers, including neural networks and genetic programs, are combined in [83]. This strategy results in an improved prediction of thyroid normal and thyroid carcinoma classes.

In this chapter genetic programming is applied to medical data widely tested in the machine learning community. More specifically, our linear variant of GP is tested on six diagnosis problems that have been taken from the Proben1 benchmark set of real-world problems [74]. The main objective here is to show that for these problems GP is able to achieve classification rates and generalization performance quite similar to NNs. The application further demonstrates the ability of genetic programming in data mining, where general descriptions of information are to be found in large real-world databases.

4.2 Benchmark Datasets

Table 4.1 gives a brief description of six diagnosis problems and the diseases that are to be predicted. For a more detailed description the reader may consult [74]. Medical diagnosis mostly describes classification tasks which are much more frequent in medicine than approximation problems.

The datasets have been taken unchanged from an existing collection of real-world benchmark problems, Proben1 [74], that has been established originally for neural networks. The results obtained with one of the fastest learning algorithms for feed-forward neural networks (RPROP) accompany the Proben1 benchmark set to serve as a direct comparison with other methods. Comparability and reproducibility of the results are facilitated

Problem	Diagnosis
cancer	benign or malignant breast tumor
diabetes	diabetes positive or negative
gene	intron-exon, exon-intron or no boundary in DNA sequence
heart	diameter of a heart vessel is reduced by more than 50% or not
horse	horse with a colic will die, survive or must be killed
thyroid	thyroid hyperfunction, hypofunction or normal function

Table 4.1: Medical diagnosis tasks of PROBEN1 benchmark datasets.

by careful documentation of the experiments. Following the benchmarking idea the results for neural networks have been adopted completely from [74]. But we verified Prechelt's results partly by own simulations. Our main objective was to realize a fair comparison between GP and NNs in medical classification and diagnosis. We will show that for all problems discussed the performance of GP in generalization comes very close to or is even better than the results documented for NNs.

All PROBEN1 datasets originate from the *UCI Machine Learning Repository* [15]. They are organized as a sequence of independent sample vectors divided into input and output values. For a better processing by neural networks the representation of the original (raw) datasets has been preprocessed in [74]. Values have been normalized, recoded, and completed. All inputs are restricted to the continuous range [0,1] except for the gene dataset which holds -1 or $+1$ only. For the outputs a binary *1-of-m encoding* is used where each bit represents one of the m possible output classes of the problem definition. Only the correct output class carries a "1" while all others carry "0". It is characteristic for medical data that they suffer from unknown attributes. In PROBEN1 most of the UCI datasets with missing inputs have been completed by 0 (30% in case of the horse dataset).

Table 4.2 gives an overview of the specific complexity of each problem expressed in the number of attributes, divided into continuous and discrete inputs, plus output classes and number of samples. Note that some attributes have been encoded into more than one input value.

Problem	#Attributes	#Inputs		#Classes	#Samples
		continuous	*discrete*		
cancer	9	9	0	2	699
diabetes	8	8	0	2	690
gene	60	0	120	3	3175
heart	13	6	29	2	303
horse	20	14	44	3	364
thyroid	21	6	15	3	7200

Table 4.2: Problem complexity of PROBEN1 medical datasets.

4.3 Experimental Setup

4.3.1 Genetic Programming

We employ the LGP approach that has been outlined in Chapter 2. For each dataset an experiment with 30 runs has been performed with LGP. Runs differ only in their choice

Parameter	Setting
Population size	5000
Number of demes	10
Migration rate (of best)	5%
Classification error weight in fitness	1
Maximum number of generations	250
Maximum program length	256
Maximum initial length	25
Crossover probability	90%
Mutation probability	90%
Instruction set	$\{+, -, \times, /, sin, e^x, if >, if \leq\}$
Register set	$\{r_0, .., r_{k-1}\}$ (k inputs)
Constant set	$\{0, .., 255\}$

Table 4.3: Parameter settings for LGP.

of a random seed. Table 4.3 lists the parameter settings used for all problems here.

For benchmarking, the partitioning of the datasets has been adopted from PROBEN1. The *training set* always includes the first 50% of all samples, the next 25% is defined as the *validation set* and the last 25% of each dataset is the *test set*. In PROBEN1 three different compositions of each dataset were prepared, each with a different order of samples. This increases the confidence that results are independent of the particular distribution into training, validation and test set.

The fitness of an individual program is always computed using the complete training set. According to the LGP algorithm described in Section 2.3 generalization performance of the best-so-far individual is checked during training by calculating its error using the validation set. The test set is used only for the individual with minimum validation error after training.

The applied fitness function $\mathcal{F}$ has two parts, a continuous component and a discrete component (see Equation 4.1). The continuous *mean square error* (MSE) is calculated by the average squared difference between the predicted output (vector) $gp(\vec{i_k})$ of an individual program gp and the desired output (vector) $\vec{o_k}$ for all n input-output samples $(\vec{i_k}, \vec{o_k})$ and $m = |\vec{o_k}|$ outputs. The discrete *mean classification error* (MCE) is computed as the average number of incorrectly classified examples.

$$
\begin{aligned}
\mathcal{F}(gp) &= \text{MSE} + w \cdot \text{MCE} \\
&= \frac{1}{n \cdot m} \sum_{k=1}^{n} (gp(\vec{i_k}) - \vec{o_k})^2 + \frac{w}{n} \cdot \text{CE}
\end{aligned}
\tag{4.1}
$$

The MCE is weighted by a parameter w. In this way, the classification performance of a program determines selection more directly while the MSE component still allows continuous fitness improvements. For fair comparison, the *winner-takes-all* classification method has been adopted from [74]. Each output class corresponds to exactly one program output. The class with the highest output value designates the response according to the 1-of-m output representation introduced in Section 4.2.

The generation in which the individual with the minimum validation error appeared defines the *effective training time*. The *classification error* of this individual on the test set characterizes the generalization performance that is of main interest here.

4.3.2 Population Structure

In evolutionary algorithms the population of individual solutions may be subdivided into multiple subpopulations. Migration of individuals among the subpopulations causes evolution to occur in the population as a whole. Wright first described this mechanism as the *island model* in biology [101] and reasoned that in semi-isolated subpopulations, called *demes*, evolution progresses faster than in a single population of equal size. This inherent acceleration of evolution by demes could be confirmed for EAs [95] and for GP in particular [94, 4]. One reason for this acceleration may be that genetic diversity is preserved better in multiple demes with a restricted migration of individuals. Diversity in turn influences the probability that the evolutionary search hits a local minimum. A local minimum in one deme might be overcome by other demes with a better search direction. A nearly linear acceleration can be achieved in evolutionary algorithms if demes are run in parallel on multi-processor architectures [4].

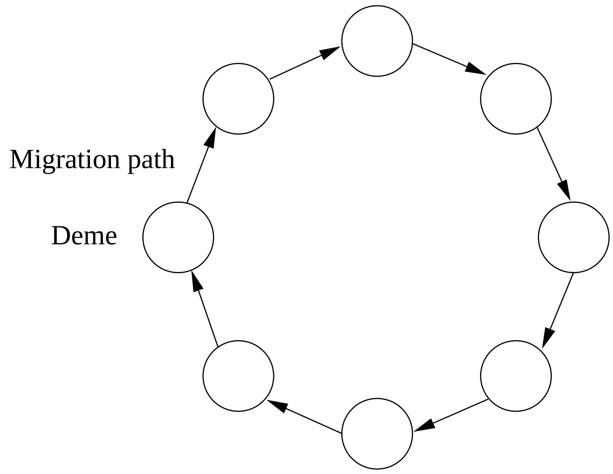

Figure 4.1: Stepping stone model of directed migration on a ring of demes.

A special form of the island model, the *stepping stone model* [47], assumes that migration of individuals is only possible between certain adjacent demes which are organized as graphs with fixed connecting links. Individuals can reach remote populations only after passing through these neighbors. In this way, the possibility that there will be an exchange of individuals between two demes depends on their distance in the graph topology. Common topologies are ring or matrix structures.

In our experiments, the population is subdivided into 10 demes each holding 500 individuals. This partitioning has been found to be sufficient for investigating the effect of multiple demes. The demes are connected by a directed ring of migration links by which every deme has exactly one successor (see Figure 4.1). After each generation a certain percentage of *best* individuals, which is determined by the *migration rate*, emigrates from each deme into the successor deme thereby replacing the worst individuals. Primarily, demes are used here to allow locally best solutions a higher reproduction by migration. By copying the best solutions of a deme into several others learning may accelerate because these individuals might further develop simultaneously in different subpopulations. In general, a more frequent reproduction of better individuals in the population increases the probability that these solutions are selected and improved. However, it may cause a premature loss of diversity, too. This negative influence is partly counteracted by the use of demes. Additionally, the migration of best is not free between demes, but restricted to certain migration paths only that are organized as a directed ring. Together with a modest migration rate this has been found to be a good compromise between faster fitness progress and preservation of diversity.

4.3.3 Neural Networks

Experimental results in [74] have been achieved using standard multi-layer perceptrons (MLPs) with fully connected layers. Different numbers of hidden units and hidden layers (one or two) have been tried before arriving at the best network architecture for each problem. The applied training method was RPROP [76], a fast and robust backpropagation variant. For further information on the RPROP parameter settings and the special network architectures the reader may consult [74].

The generalization performance on the test set is computed for the state of the network with minimum validation error. The *effective training time* of the neural network is measured in the number of epochs until this state is reached. One *epoch* is over if all training samples have been presented to the network.

4.4 Results and Comparison

4.4.1 Generalization Performance

Table 4.4 shows the classification error rates obtained with genetic programming and neural networks, respectively, for the medical datasets discussed in Section 4.2. Best and average CE of all GP runs are documented on the validation set and test set for each medical dataset, together with the standard deviation. A comparison with the test classification error of neural networks (reprinted from [74]) is the most interesting here. For that purpose the difference Δ between the average test errors of NN and GP is printed in percent of the largest value. A positive Δ indicates improved GP results over NN. A negative Δ indicates better NN results, respectively. Unfortunately, the classification results on the validation set and the results of best runs are not specified in [74] for NNs.

Our results demonstrate that LGP is able to reach a generalization performance similar to multi-layer perceptrons using the RPROP learning rule. The rather small number of runs performed for each dataset may, however, give an order of magnitude comparison only. In addition, the results for GP are not expected to rank among the best, since parameter settings have not been adjusted to each benchmark problem. This has deliberately not been carried out in order to show that even a common choice of the GP parameters can produce reasonable results. In contrast, at least the NN architecture has been adapted specifically for each dataset in [74]. Finally, the PROBEN1 datasets are prepared for being advantageous to NNs but not necessarily to GP. This is especially true for the coding of input attributes and outputs whose dimensions are larger than in the original UCI datasets (see Section 4.2). For instance, even if multiple program outputs required for a winner-takes-all classification are easy to handle in linear GP by using multiple output registers, they do not necessarily produce better results.

Notably, for the gene problem the test classification error (average and standard deviation) has been found to be much better with GP. This is another indication that GP is able to handle a very high number of inputs efficiently (see Table 4.2). On the other hand, cancer turned out to be considerably more difficult for GP than for NN judged by the percentage difference in average test error.

	GP						NN		
Problem	Validation CE (%)			Test CE (%)			Test CE (%)		Δ (%)
	best	*mean*	*std.dev.*	*best*	***mean***	*std.dev.*	***mean***	*std.dev.*	
cancer1	1.7	2.5	0.3	0.6	2.2	0.6	1.4	0.5	–36.7
cancer2	0.6	1.4	0.4	4.0	5.7	0.7	4.8	0.9	–16.6
cancer3	1.7	2.6	0.4	3.5	4.9	0.6	3.7	0.5	–24.9
diabetes1	20.3	22.2	1.1	21.4	24.0	1.4	24.1	1.9	+0.6
diabetes2	21.4	23.2	1.3	25.0	27.9	1.5	26.4	2.3	–5.1
diabetes3	25.5	26.7	0.7	19.3	23.1	1.3	22.6	2.2	–2.2
gene1	7.8	11.2	2.3	9.2	13.0	2.2	16.7	3.8	+22.2
gene2	9.1	12.9	2.3	8.5	12.0	2.2	18.4	6.9	+35.1
gene3	7.2	10.8	2.1	10.1	13.8	2.1	21.8	7.5	+36.6
heart1	7.9	10.5	2.4	18.7	21.1	2.0	20.8	1.5	–1.4
heart2	14.5	18.6	2.4	1.3	7.3	3.3	5.1	1.6	–29.8
heart3	15.8	18.8	1.5	10.7	14.0	2.0	15.4	3.2	+9.2
horse1	28.6	32.4	2.2	23.1	30.6	2.2	29.2	2.6	–4.5
horse2	29.7	34.3	2.7	31.9	36.1	2.0	35.9	2.5	–0.7
horse3	27.5	32.7	1.9	31.9	35.4	1.8	34.2	2.3	–3.6
thyroid1	0.8	1.3	0.3	1.3	1.9	0.4	2.4	0.4	+19.8
thyroid2	1.1	1.6	0.3	1.4	2.3	0.4	1.9	0.2	–17.3
thyroid3	0.9	1.5	0.2	0.9	1.9	0.4	2.3	0.3	+17.2

Table 4.4: Classification error rates of GP and NN for PROBEN1 medical datasets. NN data taken from [74]. Difference Δ in percent. Positive Δs indicates improved GP results over NN.

Looking closer, classification results for the three different datasets of each problem show that the difficulty of a problem may change significantly with the distribution of data into training, validation and test set. Especially the test error differs with the three different distributions. For instance, the test error is much smaller for dataset heart2 than for heart1. For some datasets the training, validation and test sets cover the problem data space differently, i.e., are less strongly correlated. As a result a strong difference between validation and test error might occur, as in case of cancer and heart.

Not for all problems, including diabetes, heart, and horse, the best classification results have been produced with conditional branches. This might be due to the fact that if branches are not necessary for a good solution they promote rather specialized solutions. Another reason may be the rather poor correlation of training data and generalization data here [74]. Other problems, especially gene, have worked better with branches. In general, branches have been found to have a much smaller influence on the generalization performance than on the training performance (not documented). How similar the gain in performance is, strongly depends on the correlation of training data and generalization data.

4.4.2 Effective Training Time

The effective training time specifies the number of *effective* generations or epochs, respectively, until the minimum validation error occurred. We can deduce from Tables 4.2 and 4.5 that more complex problems cause more difficulty for GP and NN and, thus, a longer effective training time. A comparison between generations and epochs is, admittedly, difficult, but it is interesting to observe that effective training time for GP shows lower variation than for NN.

	GP		NN	
Problem	effective Generations		effective Epochs	
	mean	*std.dev.*	*mean*	*std.dev.*
cancer1	26	24	95	115
cancer2	26	25	44	28
cancer3	17	11	41	17
diabetes1	23	14	117	83
diabetes2	28	25	70	26
diabetes3	21	15	164	85
gene1	77	21	101	53
gene2	90	20	250	255
gene3	86	14	199	163
heart1	17	14	30	9
heart2	20	14	18	9
heart3	21	18	11	5
horse1	18	16	13	3
horse2	19	16	18	6
horse3	15	14	14	5
thyroid1	55	18	341	280
thyroid2	64	15	388	246
thyroid3	51	14	298	223

Table 4.5: Effective training time of GP and NN (rounded).

4.4.3 Acceleration of Absolute Processing Time

Table 4.6 shows the percentage of noneffective instructions (and effective instructions) averaged over all programs of a run and over multiple runs (30 here) as identified by Algorithm 3.1 for the medical problems under consideration. The potential acceleration of runtime, that is obtained when removing these introns before each program is evaluated, directly results from the intron rates (using Equation 3.1). In general, an intron rate of **80%** has been observed which corresponds to an average decrease in runtime by the intron elimination of about a factor **5**. This speedup is of practical significance especially when operating with large datasets as they occur in medicine. A further benefit of the reduced execution time is that the effective linear genetic programs may operate more efficiently in time-critical applications. The reader may recall that the elimination of introns cannot have any influence on the fitness or classification performance (see Section 3.2.1).

From Table 4.6 it may also be concluded that the average percentages of effective program size strongly vary with the problem. The standard deviation of program size has proven to be amazingly small between single runs of the same problem, by comparison. The differences between the three datasets tested for each problem are found even smaller and are, therefore, not specified here.

Different instruction types may cause different computational costs, of course. Compared to most operations, branch instructions are rather cheap in execution time, for instance.

Problem	Introns (%)		Effective Code (%)		Speedup
	mean	*std.dev.*	*mean*	*std.dev.*	
cancer	65.5	2.8	34.6	2.8	2.9
diabetes	74.5	0.6	25.5	0.6	3.9
gene	90.5	1.1	9.5	1.1	10.5
heart	88.2	0.9	11.8	0.9	8.5
horse	90.8	0.4	9.2	0.4	10.9
thyroid	72.2	1.8	27.8	1.8	3.6

Table 4.6: Percentage of introns and effective code per run in percent of the absolute program length. Factors show speedup if only the effective code is executed. Notable differences exist between problems.

Additional computation is saved with branches because not all (conditional) operations of a program are executed for each training sample. In general, the calculation of the relative speedup factors relies on the assumption that the different components of the instruction set are approximately uniformly distributed in the population – over the effective code as well as over the noneffective code.

4.4.4 Acceleration of Effective Training Time

Another important result of our GP experiments is that effective training time can be reduced considerably by using semi-isolated subpopulations together with an elitist migration strategy (as described in Section 4.3.2). Moreover, this is possible without leading to a notable decrease in generalization performance. A comparable series of runs without demes but with the same population size has been performed for the first dataset of each problem. The average classification rates documented in Table 4.7 differ only slightly from the results obtained with a demetic population (see Table 4.4).

	GP without Demes					
Problem	Validation CE (%)			Test CE (%)		
	best	*mean*	*std.dev.*	*best*	*mean*	*std.dev.*
cancer1	1.1	2.1	0.5	1.2	2.9	1.2
diabetes1	19.3	21.4	0.7	20.3	24.4	1.7
gene1	7.7	11.0	3.0	9.0	12.6	3.1
heart1	7.9	11.0	3.0	18.7	22.3	2.9
horse1	26.4	32.4	1.9	22.0	30.7	3.5
thyroid1	0.7	1.3	0.4	1.2	2.0	0.5

Table 4.7: Classification error rates of GP without demes. Average results similar to results with demes (see Table 4.4).

Table 4.8 compares the effective training time using a panmictic (non-demetic) population with the respective results from Table 4.5 after the same maximum number of 250 generations. On average, the number of effective generations is reduced by a factor of about **3**. Thus, a significantly faster convergence of runs is achieved by using a demetic approach that allows only better individuals to migrate.

	GP *with* Demes		GP *without* Demes		
Problem	effective Generations		effective Generations		Speedup
	mean	*std.dev.*	*mean*	*std.dev.*	
cancer1	26	24	62	67	2.4
diabetes1	23	14	62	53	2.7
gene1	77	21	207	42	2.7
heart1	17	14	68	75	4.0
horse1	18	16	59	63	3.3
thyroid1	55	18	200	36	3.6

Table 4.8: Effective training time of GP with and without demes. Significant acceleration with demes and an elitist migration strategy.

4.4.5 Further Comparison

Note that reducing the (relative) training time on a generational basis affects the absolute training time, too, because runs may be stopped earlier. Comparing the absolute runtime of genetic programming and feed-forward neural networks, the fast NN learning algorithm has been found to be superior. One should keep in mind, however, that large populations have been used with the GP runs to guarantee a sufficient diversity and a sufficient number of (not too small) subpopulations. Because we concentrate on a comparison in classification performance the configuration of our LGP system has not been optimized for runtime. Nevertheless, the proposed speedup techniques for (L)GP help to reduce the difference in runtime to NN, especially if smaller populations of genetic programs are used.

In contrast to neural networks, GP is not only capable of predicting outcomes but may also provide insight into and a better understanding of the medical diagnosis by an analysis of the learned models (genetic programs) [63]. Knowledge extraction from genetic programs is more feasible with programs that are compact in size and free from redundant information. Thus, the elimination of noneffective code in our LGP system may serve another purpose in generating more intelligible results than do NNs.

4.5 Discussion and Future Research

All tested datasets originate from a set of real-world benchmark problems established and preprocessed especially for the benefit of neural networks. For genetic programming there is still a lack of a *standard set* of benchmark problems. Such a set would give researchers the opportunity for a better comparability of their published methods and results. An appropriate benchmark set should be composed of real-world datasets taken from real problem domains as well as artificial problems where the characteristics of the data are exactly known.

But a set of benchmark problems is not enough to guarantee comparability and reproducibility of results. A single parameter that is not published or an ambiguous description can make an experiment unreproducible. In order to make a direct comparison of published results easier a set of *benchmarking conventions* has to be defined, along with the benchmark problems. These conventions should describe standard ways of setting up and documenting an experiment, as well as measuring and documenting the results. A step in this direction has been taken by Prechelt for neural networks [74].

Besides, the best generalization on the validation set was reached long before the final generation. Wasted training time can be saved if runs are stopped earlier. Appropriate *stopping rules* that monitor the progress in real-world fitness and generalization over a period of generations need to be defined.

4.6 Conclusion

We reported on LGP applied to a number of medical classification tasks. It was demonstrated that, on average, genetic programming performs competitive to RPROP neural networks with respect to the generalization performance.

The runtime performance of genetic programming becomes especially important for time-critical applications or when operating with large datasets from real-world domains like medicine. Two techniques were presented that reduced the computational costs significantly.

First, the elimination of noneffective code from linear genetic programs resulted in an average decrease in runtime of about factor 5 here. Second, the number of effective generations of the evolutionary algorithm was reduced without decreasing the performance by means of a demetic population in combination with an elitist migration strategy. In doing so, the number of effective generations became remarkably small.

Chapter 5

Design of Linear Genetic Operators

Contents

Traditionally, crossover is applied in genetic programming for varying the contents and the size of programs. In this chapter we systematically introduce alternative variation operators for the linear program representation – including variation schemes that work exclusively with mutations – and compare their influence on primarily the prediction performance and the complexity of solutions.

Besides the two basic variants, *recombination-based* LGP and *mutation-based* LGP, we distinguish two different levels of variation. *Macro variations* operate on instruction level (or *macro level*). That is, an instruction represents the smallest unit. *Micro variations* happen on the level of instruction components (*micro level*) that are registers, operators, and constants. Only macro variations influence program growth. All recombination and mutation operators compared in this chapter are macro operators. Those are further subdivided into *segment variations* and *instruction variations* depending on whether an arbitrary sequence of instructions or only one instruction is allowed to be changed.

We will see that the performance of a variation operator strongly depends on its maximum (and average) step size on the symbolic program structure, its influence on code growth, and the proportions of effective variations and neutral variations. Among other things, macro mutations with a minimum step size will turn out to be most efficient if these guarantee a change of the (structurally) effective code. We also investigate how linear genetic programs may be manipulated more efficiently by respecting their functional structure.

5.1 Variation Effects

Basically, two different effects of a variation operator can be distinguished in evolutionary computation. These are its effect on the genotype and its effect on the phenotype. In GP the genotype is represented by the program structure while the phenotype is determined by the semantics (execution) of a program.

5.1.1 Semantic Variation Effects

The phenotype quality is measured by a fitness function $\mathcal{F} : \mathcal{P} \rightarrow \mathbb{R}_0^+$. Fitness distributions have been proposed as a means for understanding (semantic) variation effects in evolutionary computation. In [34] the *fitness distribution* (FD) of a variation operator v is described as the probability distribution of the offspring fitness $\mathcal{F}_o$ depending on the fitness of parent(s) $\mathcal{F}_{\{p\}}$:

$$\mathrm{FD}_v(\mathcal{F}_{\{p\}}) := \mathrm{Prob}(\mathcal{F}_o|\mathcal{F}_{\{p\}}). \tag{5.1}$$

A fitness distribution is quite complex and, in general, rather difficult to compute. In practice it is usually sufficient and even more interesting to focus on important characteristic features of the fitness distribution only [65, 42] which serve as an approximation of the actual distribution. If we assume that a better fitness always means a smaller fitness value (error) $\mathcal{F}$ the following definitions are valid.

Definition 5.1 (*constructive/destructive/neutral variation*) A variation is defined as *constructive* iff the difference between the fitness $\mathcal{F}_p$ of a parent individual and the fitness $\mathcal{F}_o$ of its offspring is positive, i.e., $\mathcal{F}_p - \mathcal{F}_o > 0$. In case of a negative difference we refer to a *destructive* variation, i.e., $\mathcal{F}_p - \mathcal{F}_o < 0$. Finally, a genetic operation is *neutral* if it does not change the fitness, i.e., $\mathcal{F}_p = \mathcal{F}_o$.

In the LGP algorithm from Section 2.3 always two offsprings are created from two parents at each iteration for comparing reasons. Either recombination is applied once between both parents and produces two offsprings or mutation is applied on each parent separately. In both cases we compare the parent and the offspring with the same index, i.e., p_1 with o_1 and p_2 with o_2. That is, the state of an individual at a certain position in the population is compared before and after it has been varied.

In the current study, we focus on the proportion of constructive, destructive, and neutral operations per generation. Such measurements regard the *direction* of semantic variation effects, but disregards other features of a fitness distribution, like the *amount* of a fitness change (see Section 5.3 below).

5.1.2 Structural Variation Effects

On the program structure we measure the proportion of so called effective and noneffective variations.

DEFINITION 5.2 (*effective/noneffective variation*) A genetic operation applied to a linear genetic program is called *effective* iff it affects the *structural* effective code after Definition 3.4. Otherwise, a variation is called *noneffective*.

Note that even if effective code is altered the program predictions for a considered set of fitness cases might be the same. An effective variation is merely meant to bring about a structural change of the effective program. There is no change of program semantics (fitness) guaranteed which is mostly due to the existence of semantic introns. It follows from the above definitions that all (structurally) noneffective variations are (semantically) neutral but not the other way around.

Measuring the *amount* of structural change between parent and offspring requires the definition of a structural distance metric between genetic programs and will be discussed in Section 5.3.

5.2 Effective Evaluation

In principle, there are two different ways to identify effective variations. Either the effectiveness is implicitly guaranteed by the genetic operator itself (see Section 5.10.4) or the effective code of an individual is compared explicitly before and after the variation (see Section 5.7.4). The latter becomes especially necessary with recombination.

By using Algorithm 3.1 the effective code of parent and offspring can be identified and extracted in linear computation time $O(n)$ with n denotes the maximum program length. In doing so, the two effective programs may be compared simply by $O(n)$ comparisons of instructions which is reduced to integer comparisons in our implementation (see Section 2.1.1). A variation is identified as effective after the comparison failed for one instruction position. Otherwise, it is noneffective by definition.

In order to avoid another application of Algorithm 3.1 to the same individual before the fitness evaluation the effective code of each program may be saved separately. A less memory-intensive alternative that is applied here marks the effective instructions within the program representation (see Section 3.2.1). An update flag for each program decides whether the effective code has already been calculated or not.

If a variation has been identified as *noneffective* the effective code is unchanged. In this case, the fitness evaluation of the offspring can be skipped since its behavior cannot be different from the parent. This produces a difference between comparing variation operators on the basis of *generations* (number of varied individuals) or *evaluations* (number of effective variations) since it is no longer guaranteed that each new (varied) individual will be evaluated, too. Evaluating individuals after effective variations only will be referred to as *effective evaluation* in the following.

Besides the removal of noneffective code before the fitness evaluation, as presented in Section 3.2.1, this is another technique to accelerate runtime of linear GP. Depending on the rate of noneffective operations that is induced by a variation operator a high amount of fitness evaluations can be saved. The overall acceleration in runtime is expressed by the factor

$$\alpha_{acc} = \frac{n_{var}}{n_{effvar}} \tag{5.2}$$

where $n_{(eff)var}$ is the number of (effective) variations.

In general, the fitness evaluation is by far the most time-consuming step in a GP algorithm. Computational costs for variation may be neglected if the time for calculating a new search point is linear in program size. Both techniques, the detection of effective variations as well as the detection of effective code, do not produce more than linear variation costs when using Algorithm 3.1.

5.3 Variation Step Size

The *variation step size* denotes the distance between a parent individual gp_p and its offspring gp_o that results from the application of one or more variation operators. The *phenotype distance* or *semantic step size* is calculated by a semantic distance metric $d_\mathcal{P} : \mathcal{P} \times \mathcal{P} \rightarrow \mathbb{R}_0^+$. The absolute difference in fitness $d_\mathcal{P}(gp_p, gp_o) := |\mathcal{F}(gp_p) - \mathcal{F}(gp_o)|$ identifies a phenotype with its fitness value which is a simplification already because the fitness function $\mathcal{F}$ is not bijective in general (see Section 1.2). However, usually much more genetic operations are destructive than constructive in GP (see below) while negative changes may become larger, on average, than positive changes. As a result, the average fitness distance $\mathcal{E}(|\mathcal{F}_p - \mathcal{F}_o|)$ is dominated by large negative outliers depending on the range of possible fitness values. To avoid this, positive and negative fitness changes may be computed separately.

Computing the *genotype distance* or *structural step size* $d_\mathcal{G}(gp_p, gp_o)$ requires an appropriate distance metric $d_\mathcal{G} : \mathcal{G} \times \mathcal{G} \rightarrow \mathbb{N}_0^+$ to be defined on the program structure. In this thesis we measure all structural step sizes absolutely in instructions, not relative to the program length. Relative step sizes are more difficult to control and to minimize during a whole run since programs grow. Moreover, the corresponding semantic step size is only partly proportional to the length of a linear genetic program.

Definition 5.3 is more precise than simply calculating the distance in program length if code is both inserted and deleted in one step, e.g., during crossover. It is also more precise than using the (average) segment length only since an exchange of code may be more destructive than a deletion or an insertion. This definition only disregards that the actual step size may be smaller due to an exchange of similar code segments at similar positions.

Definition 5.3 (*absolute structural step size*) For macro operators in linear GP let the *absolute step size* be defined as the number of instructions that are added to a linear program *plus* the number of instructions that are removed during one variation step.

Accordingly, the *effective step size* may be defined intuitively as the number of inserted and/or deleted effective instructions. When using unrestricted segment variations the effective step size may be sufficiently approximated in this way. However, such a definition is imprecise since additional instructions may become effective or noneffective above the variation point. Especially, if the absolute variation step size is minimum (instruction variation) these side-effects within the linear program structure become more relevant. In this case the following definition is more precise.

Definition 5.4 (*effective structural step size*) The *effective step size* counts instructions that are added to or removed from the *effective* program *and* depending instructions that change their effectiveness status only, i.e., that are deactivated or reactivated.

Micro mutations affect, by definition, a single instruction component only. That is, their absolute step size is always constant and minimum. Nonetheless, their effective step size may be much larger. This is the case, for instance, if an effective instruction register is replaced on which the effectiveness of many other instructions depends.

So two different structural step sizes may be distinguished in linear GP. On the functional level the absolute step size measures the total number of deleted or inserted graph nodes. The effective step size, instead, counts all instruction nodes that are connected to or disconnected from the effective graph (see Section 3.3). Thus, the effective step size observes the functional structure of a linear program better. In general, the distance between the effective code of parent and offspring is more precise because it is more closely related to the fitness distance. A smaller effective step size may be assumed to lead to a smaller change in fitness. In Chapter 8 we present distance metrics that calculate the effective distance between linear genetic programs. This information is used to control the variation step size more explicitly on the *effective* code. In this chapter the (absolute) variation step size is controlled on the *full* program structure.

The proportion of noneffective code within a linear genetic program together with the absolute program size influences the step size that is induced by segment variations, including recombination and mutation, on the effective program. The higher the intron rate is the less effective instructions are deleted and/or inserted, on average. Such an implicit control mechanism of the effective step size assumes that effective and noneffective instructions are approximately uniformly distributed in linear genetic programs.

Even if introns do not directly contribute to the fitness of a program, they increase the average fitness and survivability of their offsprings in this way. That is, an explicit or implicit reduction of effective step size increases the *effective fitness* [65] or the *evolvability* [3] of the population programs. Actually, the notion of effective step size allows the evolvability of linear genetic programs to be measured and controlled more explicitly. In doing so, the effective step size considers not only structural aspects of a genetic program, like the intron rate, but also the influence of (the absolute step size of) the variation operator. We will demonstrate in this chapter (and in Chapter 8) that a minimization of (effective) step sizes, i.e., a maximization of the effective fitness, yields the best performance.

5.4 Causality

Unless otherwise stated the term *step size* will refer to the absolute *structural* variation distance in the following. In evolutionary computation this term originates from the idea of a *fitness landscape* [59, 43] where all possible solutions of the (genotype) search space are organized in a structural neighborhood – by using a structural distance metric – and their fitness values constitute a more-or-less smooth surface. The application of a variation operator corresponds to performing one step on the fitness landscape. Both the roughness of the surface and the step size of the variation operator determine the success of the evolutionary search process.

On the one hand, the variation operator has to allow progress in steps that are small enough, on average, to approach a global optimum solution or at least a good local optimum adaptively. That means, in other words, to *exploit* the fitness information of adjacent search points by a gradient descent. One strength of evolutionary algorithms is that this gradient is not followed exactly, but rather by a *gradient diffusion* [75]. Due to the fact that new search points are selected randomly without a certain direction, an evolutionary search will less likely get stuck in local minima (suboptima) of the fitness landscape. Usually there is more than one global optimum (in the genotype space) since programs with optimum fitness are not necessarily unique by structure. In GP this is already true because of the redundant code in programs.

On the other hand, the average variation step size must not be too small. Otherwise the global evolutionary progress may be too much restricted. Additionally, a sufficient proportion of larger steps may be required to avoid that the evolutionary process gets stuck early in a local suboptimum. That is, a sufficient *exploration* of the fitness landscape has to be maintained. This may depend, however, on other factors like the population size and the diversity of the population material, too. Moreover, exploration is depending on a sufficient proportion of neutral variations, which allow neutral walks over the fitness landscape.

This chapter will show that linear genetic programming profits strongly from a reduction of variation step size. This might be interpreted in such a way that an *exploration-exploitation trade-off* does not exist. Even minimum step sizes on the program structure seem to be still large enough to escape from local minima.[1] One reason is that the fitness landscape is not perfectly smooth, especially when operating on a symbolic level (genetic programs). Even smallest changes of the program structure may still result in large changes of program semantics.

Strong causality requires a completely “smooth” fitness landscape [75]. Therefore, this feature postulates Equation 5.3 to be valid for any three search points:

$$\forall p_1, p_2, p_3 \in \mathcal{G} : d_{\mathcal{G}}(p_1, p_2) \leq d_{\mathcal{G}}(p_1, p_3) \Leftrightarrow d_{\mathcal{P}}(p_1, p_2) \leq d_{\mathcal{P}}(p_1, p_3) \tag{5.3}$$

That is, small changes of position (individual) in the high-dimensional landscape always imply small changes in height (fitness). Strong causality is, however, not a necessary condition for the function of evolutionary algorithms in general. Actually, this condition is not strictly fulfilled by most evolutionary algorithms. Already from observations in nature we may not assume a strong causality between genotype and phenotype. In biological evolution the DNA may be subject to strong modifications without affecting the organism significantly. On the other hand, larger modifications of the phenotype may result from only little genotype changes. Nevertheless, the vast majority of natural variations on genotype level is rather small and is expressed (if ever) in small variations of the phenotype. Among other things, this is due to the redundancy of the genetic code that comes from intron segments by which many mutations stay neutral or nearly neutral.

Nevertheless, a fitness landscape must be smooth at least in local regions (*locally strong causality*) [75, 81]. Otherwise, evolutionary search may not be more powerful than random search. In an extremely rugged surface a search point (individual) contains only little or no information about the expected fitness of its direct neighbors. Besides ruggedness of

[1] The fitness function always minimizes a prediction error in this thesis.

the fitness landscape, flat regions, where neighboring points have the same fitness, make a problem hard to be solved by an evolutionary algorithm. On such *fitness plateaus* no gradient information is available. Contrary to this, hills and valleys in the fitness landscape represent regions with a gradient information, i.e., *local maxima* and *local minima*.

In GP the surface of the fitness landscape depends not only on the problem but on the system configuration, too, especially the provided program instructions. Neutral variations are important if a problem constitutes wide fitness plateaus. These occur especially with discrete fitness functions. The existence of intron code makes neural variations more likely, too, especially if variation step sizes are small. In flat regions of the fitness landscape neutral variations maintain evolutionary progress by a random exploration in the genotype space. That is, the population spreads wider over a fitness plateau by a neutral drift which increases the probability to find a better suboptimum. If a positive fitness gradient has been found the population may concentrate on this local optimum again, i.e., individuals that are more successful than others for that region will spread faster in the population.

Changing a small program component in genetic programming may lead to almost arbitrary changes in program behavior. On average, however, we may assume that the less instructions are modified the smaller the fitness change will be. That is, with a high probability smaller variations in genotype space, i.e., smaller variation step sizes, result in smaller variations in phenotype space, i.e., smaller fitness distances. Such a *stochastic causality* is a necessary precondition of a program representation and its genetic operators. In Section 8.7.1 a positive correlation between structural and semantic step sizes will be shown experimentally for different variation operators and problems.

5.5 Selection of Variation Points

Due to the hierarchy of nodes in tree programs a variation point (node) can be expected to be more influential the closer it lies to the root. If nodes are selected independent from their position deeper nodes are automatically chosen more frequently because most nodes are closer to a leaf. In a completely balanced binary tree of n nodes exactly $\lfloor \frac{n}{2} \rfloor$ nodes are inner nodes and $\lceil \frac{n}{2} \rceil$ nodes are leafs. Thus, half of the variation points would fall upon constants or variables. This implicit bias of tree crossover results in a lower variation probability and, thus, in a loss of diversity in tree regions closer to the root. In order to compensate this tendency Koza [51] imposes an explicit counter bias on the crossover operator by selecting inner (function) nodes with a high probability (90 percent). An alternative is to select the depth first and then select a variation point among all nodes of that depth with the same probability [37].

In a linear program the situation is different. One may assume that each program position has a more similar influence on program semantics, at least if a rather moderate number of registers is provided. Recall that the internal structure of an LGP program, as defined in Chapter 3, represents a directed acyclic graph (DAG) that is restricted in width through the number of provided registers (see Section 3.3). While in a tree each node is reached via a unique path from the root, i.e., it is connected to only one incoming edge, in a DAG more than one program path may lead to the same node, i.e., a node may be connected to several incoming edges. Therefore, it may be justified to select each instruction (variation point) with the same probability during variation.

However, even if the *maximum* width of the graph representation is restricted and the number of incoming edges is free in principle, this does not provide enough information about the specific functional structure of a certain linear program. The algorithms that have been presented in Section 3.4 extract special features about the functional or imperative program structure. Among other things, this information may be used to bias the choice of variation points more precisely.

In Section 5.11.6 mutation points will be selected for different probability distributions depending on its effective position in the imperative representation. The relative position of an effective instruction in the (effective) program is of minor importance as long as all instructions are selected for the same probability. Only if the selection of variation points is non-uniform, e.g., biased towards the end or the beginning of the imperative program, it may become important that at least approximately the relative position of an instruction is similar to the position of its corresponding node in the functional program. A small average effective dependence distance, for instance, indicates that the order of instructions is high, i.e., functionally dependant instructions lie close to each other in the imperative code.

Furthermore, it may be promising to select an instruction position for mutation depending on its degree of effectiveness or on the number of effective registers. The more effective instructions depend on the mutated one the higher is the expected effective variation step size, i.e., the more instructions may be deactivated. In Section 5.10.5 we will discuss mutation operators that use these structural features to minimize the effective mutation step size by selecting the mutation point accordingly.

5.6 Suggested Properties of Variation Operators

Together with the selection operator, the variation operators determine the efficiency of an EA and its representation of individuals. Before we discuss and compare various genetic operators for the linear program representation in particular, we summarize some general properties of variation operators and program representation in this section that we believe are especially important for genetic programming. The following general rules are meant to be independent from a special type of program representation. Some design rules are also valid for evolutionary algorithms in general (see, e.g., [98]).

(1) First of all, genetic programming is working with a variable length representation that is supposed to grow during the course of a run. It is a common practice to start the evolutionary process with relatively small initial programs. Usually fitter solutions require a certain minimum complexity, i.e., are located in more complex regions of the search space. The variation operator(s) must provide for a *sufficient growth* of programs within an observed period of generations, together with the selection operator that favors longer programs if they show a better performance.

(2) Another important property is *local search*. That means a variation operator (or a combination of variation operators) should explore the region around the parent search point(s) more intensively than more distant regions of the search space. This implies that the structural similarity between parent and offspring should be higher, on average, than between arbitrary individuals. If we assume a fitness landscape to be smooth at least in local regions, good search points are at least partly surrounded by other good search points. From these points small variation steps allow a more precise and continuous approximation to better solutions.

(3) We recommend the use of *minimum step sizes* on the (symbolic) program structure. The smallest GP operations that change program size, too, are the insertion or the deletion of a single instruction node. Usually even smallest variations of a program structure induce sufficiently large semantic steps, as discussed in Section 5.4.

(4) A specific design of efficient genetic operators in evolutionary computation strongly depends on the representation of individuals. The phenotype function and, thus, the

fitness, should be efficiently computable from the genotype representation (*efficient interpretation*) to keep the time of fitness evaluation as short as possible. Moreover, the genotype representation should allow efficient variations. In both cases, the computation time should be linear in the program size.

(5) The program representation should offer a sufficient freedom of variation (*high variability*) to allow small structural variations at each program position and throughout the whole run. Besides, it may be advantageous if noneffective code may emerge at each position with about the same probability.

(6) In order to guarantee that all effects on a program are reversible each genetic operator should be applied together with its inverse operator (*reversibility*). Additionally, it may be postulated that two inverse genetic operations should happen with the same probability (*symmetry*), i.e., without any bias towards a certain search direction. Then the search direction is only determined by selection. This, however, may contradict a sufficient program growth. Especially if the minimum step size rule is applied, an explicit grow bias in the macro operator has been found advantageous (see below).

(7) If not stated otherwise, all variation operators in this thesis are *bias-free*, i.e., would not let programs grow *without* fitness selection. That is, code growth does not occur just by the influence of genetic operators. In Chapter 9 we will analyse implicit biases that exist only in the presence of fitness information.

(8) Program solutions produced by a variation operator in GP must be valid in terms of the underlying programming language, i.e., they must satisfy constraints of the program structure. This property has been referred to by Koza as *syntactic closure*. The feasibility of a program solution, in general, may either be guaranteed implicitly by the variation operators or, if this is not possible, in a post-processing step by special repair mechanisms.

(9) In most program representations used in GP redundant code parts can be identified that do not contribute to its phenotype function. In general, too large solutions are more inflexible during the evolutionary process and may increase evaluation time. Unnecessary program growth in genetic programming has become known as the *bloat effect* (see also Chapter 9). In order to avoid these problems variation operators are required to keep the rate of redundant code as small as possible (*minimum code redundancy*). Note that a lower code redundancy reduces the genotype search space allowing the genotype-phenotype mapping to become more injective.

(10) A high proportion of redundant code in programs reduces the effectiveness of genetic operations. The more intron code has emerged the higher is the probability that later variations will not change the effective code at all. The same is true for a small (maximum) variation step size. As a result, evolution may progress slower within a certain number of generations. Provided that redundant code elements may be detected efficiently for a representation, variation may be concentrated on the remaining more effective code. A *high effectiveness* of genetic operations may be supposed to reduce the proportion of (useless) neutral variations. Note that neutral variations are important to a certain extent only. Since neutral variations perform random walks on the fitness landscape, most steps may be expected to be useless while only a small fraction may be progressive (on the code level). We will see that this is especially true in LGP if many variations change or create structural introns.

5.7 Segment Variations

In this section we investigate *segment variations*, i.e., macro variations that delete and/or insert instruction segments whose length is normally restricted only by the program length. Different recombination and mutation operators are discussed for the linear program representation. In particular, this includes the standard variant of linear genetic programming which applies linear crossover.

5.7.1 Linear Crossover

As already described in Section 2.3.4, the standard linear crossover operator always produces two offsprings by exchanging two arbitrarily long, contiguous subsequences (*segments*) of instructions between two parent individuals. This principle has been illustrated in Figure 2.4. By definition, linear crossover guarantees a minimum segment length of one instruction (= minimum program length l_{min}). The implementation of linear crossover as applied in this thesis is described by Algorithm 5.1. In the following we use identifier cross to refer to this operator. The maximum length of segments sl_{max} is unrestricted, i.e., it equals the program length. That is, the whole program code may be replaced in one genetic operation. Let the term *crossover point* always denote the *first* instruction position of a segment. The end of a segment is uniquely identified by the segment length. The position of the first instruction in program is always 0.

Algorithm 5.1 (*linear crossover*)
Parameters: two linear programs gp_1 and gp_2; minimum and maximum program length l_{min}, l_{max}; maximum segment length sl_{max}; maximum distance of crossover points d_{max}; maximum difference in segment length sd_{max}.

1. Randomly select an instruction position i_k (crossover point) in program gp_k ($k \in \{1, 2\}$) with distance $|i_1 - i_2| \leq min(l(gp_1) - 1, d_{max})$ and length $l(gp_1) \leq l(gp_2)$.
2. Select an instruction segment s_k starting at position i_k with length $1 \leq l(s_k) \leq min(l(gp_k) - i_k, sl_{max})$.
3. While difference in segment length $|l(s_1) - l(s_2)| > sd_{max}$ reselect segment length $l(s_2)$.
4. Assure $l(s_1) \leq l(s_2)$.
5. If $l(gp_2) - (l(s_2) - l(s_1)) < l_{min}$ or $l(gp_1) + (l(s_2) - l(s_1)) > l_{max}$ then
   (a) Select $l(s_2) := l(s_1)$ or $l(s_1) := l(s_2)$ with equal probabilities.
   (b) If $i_1 + l(s_1) > l(gp_1)$ then $l(s_1) := l(s_2) := l(gp_1) - i_1$.
6. Exchange segment s_1 in program gp_1 by s_2 from program gp_2 and vice versa.

If the crossover operation cannot be executed because one offspring would exceed the maximum program length, equally long segments are exchanged. Algorithm 5.1 selects randomly one of the two segment lengths in this case. Due to the fact that the crossover points are selected *before* the segment lengths in Step 1, the algorithm is biased towards selecting shorter segments more frequently. Instead, the selection of crossover points is unbiased, i.e., their distribution is uniform. Experimental results will show below that a restriction of the segment length is much less critical than restricting the free choice of crossover points. An alternative crossover implementation might select the segment lengths first for the same probability as the variation points.

It is important to note that linear crossover, in general, is not explicitly biased towards creating larger programs already on its own. Because it only *moves* code within the population and because crossover points are selected randomly, the average program length is not growing without fitness selection.

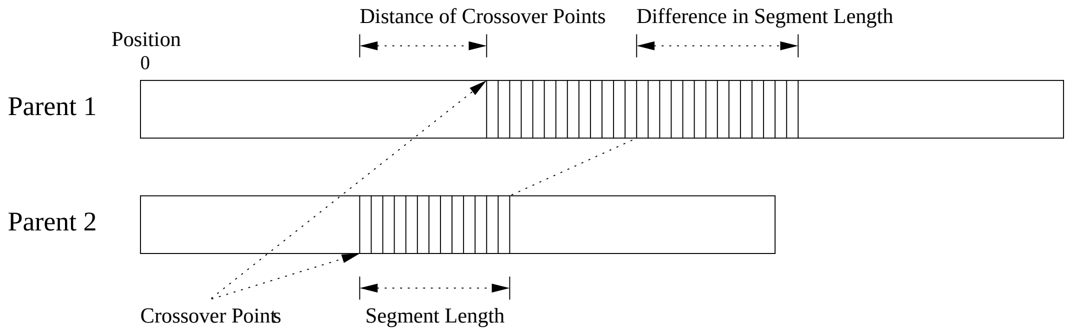

Figure 5.1: Basic parameters of linear crossover.

One way to reduce the structural step size of linear crossover more explicitly is by a maximum limit on the segment length. A *relative* upper bound for the segment length in percent of the current program length is not a feasible alternative. First, the segment length would still depend on the absolute program length. Because programs grow during a run such relative step sizes would increase, too. Second, in a linear genome the influence of the segment length on program semantics is partly independent from the program length. That is, the influence of a certain amount of varied code may be lower in a longer program only to a certain extent (see Section 9.8.6).

Another crossover parameter besides the *maximum segment length* is the *maximum distance of crossover points* d_{max} (in instructions) between both parents. A restriction of this distance reduces the probability that a piece of code may migrate to another program position by variation which necessarily implies a restriction of variation freedom.

As a third parameter that influences the performance of crossover the *maximum difference in segment length* sd_{max} between parents may be restricted. This difference controls the average step size of linear crossover together with the absolute segment length. If $sd_{max} := 0$ no program growth is possible. By setting sd_{max} to a moderate value a simple *size fair crossover* is realized in linear GP. Such an operator is more complicated to realise with subtree crossover [56].

In Figure 5.1 an illustration of these three control parameters is given for a better understanding. Besides, the performance of linear crossover might be influenced over the probability distributions of crossover points, segment lengths, or length differences. For instance, segment lengths may either be selected uniformly distributed over a maximum range (standard case) or normally distributed such that smaller or larger segments are exchanged more frequently.

Obviously, there is an analogy between crossover of DNA strings in nature and crossover of instruction sequences in linear GP. In fact, this analogy to biological crossover was the original inspiration for the use of crossover in evolutionary algorithms. On the other hand, there are some basic differences, too. The vast majority of crossover operation in nature is homologous. Biology causes homology through a strict base pairing of equally long DNA sequences while similarity of structure is closely related to similarity of (gene) function.

Nordin *et al.* [69] propose the use of *homologous crossover* in (linear) GP. The basic idea is that more similar sequences of instructions are exchanged during the course of evolution which may also be regarded as an indirect reduction of crossover step size. It has to be noted, however, that homologous linear crossover implies a restriction of both the distance of crossover points *and* the difference in segment length. We will demonstrate in Section 5.9.4 why a limitations of both parameters may not always be advantageous.

In nature base information of a DNA string – coding a certain type of protein – has to be much more place bound than this is necessary for instructions of a genetic program. Otherwise, a high survival rate of offsprings cannot be guaranteed. In GP there is no equivalent criterion for viability. Even programs with a relatively poor fitness may still pass on their information even if this happens with a lower probability. Usually the full range of fitness values is regarded without any given minimum.

5.7.2 One-Point Crossover

Standard linear crossover may also be regarded as a *two-point crossover* because the end of an exchanged instruction segment is variable, too. That is, it may be located in the midst of a parent program. With a *one-point crossover* (abbr. onepoint) programs are crossed at one point only. That is, the end of the crossed code segment is always identical to the end of program (see Algorithm 5.2). If a new individual would exceed the maximum program length the two crossover points are chosen at equal positions in both parents. Compared to two-point crossover, one-point crossover necessarily leads to larger absolute step sizes since larger segments of instructions are exchanged, on average. Additionally, the absolute step size may not be restricted that easily by a control parameter as this is possible with the standard operator.

ALGORITHM 5.2 (*one-point crossover*)
Parameters: two linear programs gp_1 and gp_2; minimum and maximum program length l_{min}, l_{max}; maximum distance of crossover points d_{max}.

1. Randomly select an instruction position i_k (crossover point) in program gp_k ($k \in \{1, 2\}$) with distance $|i_1 - i_2| \leq min(l(gp_1) - 1, d_{max})$ and length $l(gp_1) \leq l(gp_2)$.
2. $l(s_1) := l(gp_1) - i_1$,
   $l(s_2) := l(gp_2) - i_2$.
3. Assure $l(s_1) \leq l(s_2)$.
4. If $l(gp_2) - (l(s_2) - l(s_1)) < l_{min}$ or $l(gp_1) + (l(s_2) - l(s_1)) > l_{max}$ then
   (a) If $l(gp_1) \geq l(gp_2)$ then $i_1 := i_2$ else $i_2 := i_1$.
   (b) Go to $\rightarrow$ 2.
5. Exchange segment s_1 in program gp_1 by s_2 from programs gp_2 and vice versa.

5.7.3 One-Segment Recombination

Crossover requires, by definition, that information is *exchanged* between individual programs. However, an exchange always includes two operations on each parent individual at the same time, a *deletion* and an *insertion* of a subprogram. The imperative program representation allows instructions to be deleted without replacement since the instruction operands, e.g., register pointers, are always defined. Moreover, instructions may be inserted at any position without a preceding deletion, at least if the maximum program length is not exceeded. Thus, if we want linear crossover to be less destructive it may be recommended, first, to execute only one operation per parent.

These considerations motivate an *one-way* or *one-segment recombination* (abbr. oneseg) of linear genetic programs as described by Algorithm 5.3. Accordingly, standard linear crossover may also be referred to as *two-segment recombination* for a better distinction. One-segment recombination may reduce the variation step size in terms of Definition 5.3. It has to be noted, however, that the actual step size of two-segment recombination may be reduced by an exchange of similar segments.

ALGORITHM 5.3 (*one-segment recombination*)
Parameters: two linear programs gp_1 and gp_2; insertion rate p_{ins}; deletion rate p_{del}; maximum program length l_{max}; minimum program length l_{min}; maximum segment length sl_{max}.

1. Randomly select recombination type *insertion* | *deletion* for probability $p_{ins} \mid p_{del}$ and $p_{ins} + p_{del} = 1$.
2. If $l(gp_1) < l_{max}$ and (*insertion* or $l(gp_1) = l_{min}$):
   (a) Randomly select an instruction position i in program gp_1.
   (b) Randomly select an instruction segment s from program gp_2 with length $1 \leq l(s) \leq min(l(gp_2), sl_{max})$.
   (c) If $l(gp_1) + l(s) > l_{max}$ then reselect segment s with length $l(s) := l_{max} - l(gp_1)$
   (d) Insert a copy of segment s in program gp_1 at position i.
3. If $l(gp_1) > l_{min}$ and (*deletion* or $l(gp_1) = l_{max}$):
   (a) Randomly select an instruction segment s from program gp_1 with length $1 \leq l(s) \leq min(l(gp_2), sl_{max})$.
   (b) If $l(gp_1) - l(s) < l_{min}$ then reselect segment s with length $l(s) := l(gp_1) - l_{min}$
   (c) Delete segment s from program gp_1.
4. Repeat steps 1. to 3. with exchanged program identifiers gp_1 and gp_2.

In traditional GP an exchange of subtrees during crossover is necessary because the constraints of the tree structure require removed code to be replaced. Nevertheless, pure deletions or insertions of subtrees may be implemented in the following manner: A deleted subtree is substituted by one of its subtrees. Likewise, a subtree is inserted at a random position such that the deleted subtree becomes a leaf of the inserted one.

If a segment (subprogram) is deleted from a parent or if a segment is inserted from another parent depends on the two probability parameters p_{del} and p_{ins}. These allow a *grow bias* or a *shrink bias* to be adjusted for one-segment recombination depending on whether $p_{ins} > p_{del}$ or $p_{ins} < p_{del}$ is true. Such an *explicit bias* allows programs to grow without fitness information. Note that such an explicit bias may not be realized with crossover because it does not alter the average program length in the population. Only two-segment mutations (see next section) allow a more frequent exchange of smaller segment by larger ones (or vice versa).

An explicit tendency for code growth from side of the genetic operator might not be necessary if the maximum segment length is unrestricted. In this case programs may grow quickly in only a few generations. A shrinking tendency may have a positive influence on the prediction quality mostly due to a reduction of code growth that indirectly reduces the absolute step size. However, restricting program growth over the maximum segment length allows a more precise control of recombination steps. In the standard configuration one-segment recombination is applied without an explicit bias, i.e., $p_{ins} = p_{del}$.

5.7.4 Effective Recombination

In principle, there are two possibilities to increase the number of effective variations and, thus, to reduce the probability that a variation stays neutral in terms of a fitness change. Either the noneffective code is reduced actively or genetic operations concentrate on the effective part more intensively.

To demonstrate that the noneffective code controls the influence of recombination on the effective code, we may remove all noneffective instructions immediately after each variation from the individuals in the population (using Algorithm 3.1). Then it has to be explicitly guaranteed that the absolute program length does not fall below the minimum (one instruction). In contrast to removing the structural introns only before the fitness calculation, the population comprises only *effective programs* and each variation automatically becomes effective. Due to the absence of noneffective instructions, variations are expected to be more destructive on the effective code. We will see, however, in Section 5.7 that the *proportions* of effective variations and destructive variations are not much affected when using linear crossover because both are already quite high when the noneffective code is included. But a higher *amount* of (structurally) effective code is modified, i.e., the average effective step size is increased. We will refer to this approach as one variant of *effective recombination* or *effective crossover* (abbr. effcross).

Some researchers [84] proposed to remove redundant code parts before tree crossover to reduce code growth. Other researchers [16] reduce the rate of neutral crossover operations by avoiding that a crossover point falls upon an intron subtree. However, it may be pointed out again that intron detection in tree-based GP is limited. Since only semantic introns exist a detection can only be accomplished incompletely and strongly depends on the problem and the provided sets of functions and terminals. In [84] unfulfilled if-statements are partly identified in tree programs and extracted.

An alternative variant of effective recombination can be realized by an explicit control of effectiveness. That means a variation is repeated until the effective code has been altered. The effective code of two programs can be compared efficiently. Prior to that Algorithm 3.1 has to be applied to calculate the effective code of the new programs. This approach does not affect the effective variation step size but may only increase the rate of effective variations.

The effectiveness of crossover operations may already be guaranteed, too, by selecting segments (for deletion) that hold at least one effective instruction (effdel). This variant does not avoid exchanges of segments that are effectively identical. Such identical exchanges become less likely, however, if the average segment length is large. It is usually not required to check after segment deletions (by means of Algorithm 3.1) if the effective code has been changed as this is necessary after segment insertions. Remember that the effectiveness status – effective or noneffective – of each instruction is logged in programs.

5.7.5 Segment Mutations

Recombination always produces two offsprings with the implementations described above. In order to guarantee that the rate of new individuals is the same for mutations always two parent individuals are selected in our LGP Algorithm 2.1.

One-segment recombination as described by Algorithm 5.3 may be modified to serve for two variants of macro mutations. *One-segment mutations* may be realized by the insertion of a randomly created subsequence s of $l(s)$ instructions in Step 2.(d) of Algorithm 5.3. In doing so, the maximum *length* of an inserted segment (as well as a deleted segment) is restricted by the length of the other parent individual which guarantees that the mutation operator is free from an explicitly length bias. That is, the average inflow of code into the population is not larger than the outflow.

Effective segment mutations insert a fully effective segment, accordingly, which is achieved by inserting $l(s)$ effective instructions successively at a position i as will be described in Section 5.10.4. On a functional level an effective segment does not necessarily form a single contiguous component for itself even if all segment instructions are connected to the effective graph component.

Deleted segments are not fully effective, but may still contain noneffective instructions. As a result, the proportion of noneffective code is explicitly reduced. Additionally, it may be guaranteed that deletions of segments are always effective, i.e., that the deleted segment includes one effective instruction at least. This, however, has been found to make a difference only if the maximum segment length is restricted to a few instructions (see Section 5.10.4).

(Effective) segment mutation may also be realized by means of the real crossover operator from Section 5.7.1. The only difference is that random segments *replace* existing segments (of any size) here. In the following the four different variants of segment mutations will be referred to as onesegmut, effonesegmut, segmut and effsegmut.

In general, it is guaranteed for each genetic operator that there is a structural variation of the program code. Only an *exchange* of code may lead to identical parents and offsprings. Insertion *or* deletion of instructions are always changing a program, instead. Only with two-segment variations, not with one-segment variations an exchange of code is practiced.

Identical exchanges are much more likely with crossover than with two-segment mutation because in the first case the possible genetic material is more restricted (to the population contents). Avoiding identical exchanges does not necessarily require repeated applications of a variation operator until the code has changed – including explicit comparisons of the program structure. It is already sufficient to select the segment lengths differently in both parents during a crossover operation. If crossover is applied with equally long segments (after the maximum length has been exceeded) the crossover points may be set at different positions, instead. However, avoiding or not avoiding identical replacements during macro variations has not been found to produce significantly different results if the segment length (and, thus, the variation step size) is large, on average.

5.7.6 Explicit Introns

As noted above, the ratio of noneffective instructions in programs controls the influence of segment variations on the effective part of code. Also because of the maximum limitation of program length this implicit control of effective step size may not prove sufficient. One problem of the inactive instructions is that they are easily reactivated when transferred from one individual into another. The effectiveness of inserted instructions depends strongly on the context in the new program and the position at which they are inserted. Both may very likely be totally different from the original program. Thus, the protective effect of the noneffective code is more a probabilistic one.

One possibility to overcome this problem offer special program elements that already represent intron code for themselves. Nordin *et al.* [67] introduced the idea of explicitly defined introns (EDIs) into (linear) GP. This stand-alone intron code does not depend on a special semantic or structural program context. Explicit introns are supposed to suppress the emergence of implicit introns in the course of a run by replacing them. In this way, they reduce the absolute program size which includes only the operational (non-EDI) instructions. In the presence of explicit introns there is less need for inducing implicit introns code. Explicit introns are not only easier to be implemented during evolution but are less brittle during manipulation by the genetic operators, too.

The higher proportion of noneffective code that occurs especially with crossover, indirectly increases the size of effective code, too. Obviously, the more code is inactive the higher is the probability for reactivations during a genetic operation. Thus, the more programs grow the more difficult it becomes to maintain a small effective code. If implicit introns are sufficiently replaced by context-independent explicit introns, however, we may hope that also a smaller (proportion of) effective code is possible.

In [67] explicitly defined introns have been implemented by a separator that is held between all coding instructions in a linear program. The non-coding separators just include a mutable integer value n which represents a “virtual” sequence of n wildcards or *empty* instructions. During crossover the EDI value between two working instructions determines the probability that the crossover point falls between them. Actually, crossover behaves just as if EDIs were real empty instructions. After crossover has been performed the EDI values at the crossover points are updated accordingly in the offspring programs.

A different realization of explicit introns is practiced in [82] for tree-based GP by including a special EDI function into the function set. Such a function ignores all its arguments except one which is returned unaltered. This is necessary since the tree program structure requires that every operational node returns a value. All ignored subtrees become inactive code, too, but may be reactivated after a crossover operation or if the EDI function is replaced by an effective function. Such explicit introns act similar to branches that hold a condition which is always wrong.

We investigate explicit introns here for linear GP in a simpler form as used in [67]. In our approach an EDI comprises a single empty instruction only and is “physically” evolved within the imperative programs. The *empty instructions* neither perform an operation on the register set nor manipulate the contents of registers. By definition, an empty instruction is not allowed to be changed. Neither it can be reactivated nor can a working instruction be transformed into an empty one. This requires a mutation operator that is restricted to coding instructions only. During the initialization a certain percentage of empty instructions is seeded into the population in addition to the coding instructions. In this way, it is guaranteed that only crossover determines how the proportion of EDIs develops in the population during a run. One may refer to this type of introns as *imperative EDIs* since they are defined on imperative level only and have no equivalent on the functional level.

Alternatively, *functional EDIs* might be implemented in linear GP as instructions holding a *non-operator* that assigns the contents of one operand to the destination register and ignores the other one (if existent). Functional EDIs deactivate instructions which depend on an ignored register operand. Such introns may become active again if the non-operator is exchanged by mutation or if the whole EDI instruction is removed by crossover. Therefore, such explicit introns hardly provide a higher functionality than the (structural) introns that occur already implicitly in linear genetic programs.

5.7.7 Building Block or Macro Mutation ?

In comparison with mutation the success of recombination, in general, depends more strongly on the composition of the population. This is true because innovation through recombination can only result from a rearrangement of the genetic material that is already existing in the population. Innovation through mutation, instead, comes from seeding new random information from outside of the population.

Originally, the crossover operator has been introduced and intensively applied in genetic programming for the claim that recombination-based search is more successful and faster than variation that is just based on random mutations. This requires that GP individuals are composed of *building blocks* [51]. In principle, a building block may be any coherent fraction of program code, i.e., an arbitrary subtree in tree-based GP or a subsequence of instructions in linear GP. The *building block hypothesis* for general evolutionary algorithms has been adopted from genetic algorithms [32, 40] and says that smaller substructures of individuals with a high positive effect on fitness are (re)combined via crossover to produce offsprings with a higher fitness potential. Thus, an individual with good building blocks has not only a better fitness but may also produce better offsprings with a higher probability by passing on its good building blocks. Advantageous building blocks are believed to spread within the population since the individuals containing them are more likely selected for variation or reproduction. Also because of its vague formulation the hypothesis may be criticized. One point of criticism is that the building block hypothesis assumes that evolutionary algorithms decompose a problem automatically into subproblems and develop global solutions by recomposing the subsolutions. But this requires the building blocks (subprograms) to be relatively independent from each other and to have a more independent (additive) influence on the fitness. That is, the fitness function has to be at least partly *separable*. Especially in GP, however, this may hardly be assumed for each combination of program representation, recombination operator, instruction set and fitness function. Another point of criticism is that the building block hypothesis does not explain the functionality and ability of (recombination-based) evolutionary algorithms to solve problems with a highly unrelated fitness function, e.g., problem configurations where most changes of the representation are neutral in terms of a fitness change.

In GP the fitness advantage or disadvantage of a certain subtree or subsequence of instructions strongly depends on its position within an individual. In addition to this, the usually complex interaction of registers in linear GP reduces the possibility that a subprogram may behave similar in another program context. Depending on the number of available registers as well as the length of the subsequence this would require many nodes to be reconnected appropriately in the functional graph equivalent. Actually, reactivations and deactivations of instructions may easily destroy the functionality of building blocks.

If the building block hypothesis is not valid recombination acts as nothing else than a macro mutation that is restricted to the genetic material in the population. However, even if the building block hypothesis is true for a certain recombination-based GP approach, a pure mutation-based approach may exist that performs better. Note that the question of whether recombination or mutation is more powerful depends on criteria like the (average) variation step size and the degree of innovation that are induced by a genetic operator. Finally, its contribution to the growth of code is important, too.

In traditional genetic programming as initiated by Koza [51] crossover is applied for the majority of variations. The role of mutations is considered of minor importance. Mutations are used for a relatively low probability only to regularly introduce some new genetic material into the population. Later, other researcher have demonstrated that mutation operators may perform better or at least as powerful as tree crossover. Angeline [7] compares normal crossover with a "crossover" operator where one parent individual is created randomly. These *subtree mutations* work mechanically similar to crossover what allows a fair comparison. From the competitive performance of subtree mutations Angeline concluded that subtree crossover is more accurately described as a macro mutation that uses material from the population only, rather than following the principle of the building block hypothesis. Other comparisons of subtree crossover and mutations in tree-based GP [37, 58, 24] report on similar results.

In general, it may be concluded that mutation-based variation and crossover-based variation in tree-based GP either have been found competitive or one approach was only slightly more successful. In principle, macro mutation operators are based on the replacement of an existing subtree by a random one at a certain variation point that is the root node of the subtree. By rearranging genetic material within the population only the crossover operator (if bias-free) implicitly guarantees that the average program length stays unchanged. When implementing subtree mutations this has to be guaranteed explicitly by inserting subtrees that are, on average, of the same size as the deleted ones. In Section 5.9.2 we will compare recombination and segment mutations in linear GP.

5.8 Experimental Setup

5.8.1 Benchmark Problems

The different variation operators and experiments discussed in this chapter are compared by using some or all of the following benchmark problems. Basically, we concentrate on (symbolic) regression and classification tasks here. Most real-world applications may be supposed to belong to one of these problem classes.

$$f_{mexicanhat}(x,y) = \left(1 - \frac{x^2}{4} - \frac{y^2}{4}\right) \times e^{\left(-\frac{x^2}{8} - \frac{y^2}{8}\right)} \tag{5.4}$$

The first problem task requires a *surface reconstruction* from a set of data points. The surface is given by the two-dimensional *mexican hat* function here (see Equation 5.4). Figure 5.2 shows a three-dimensional plot of the function visualizing the surface that has to be approximated.

$$f_{distance}(x_1, y_1, .., y_n, y_n) = \sqrt{(x_1 - y_1)^2 + .. + (x_n - y_n)^2} \tag{5.5}$$

The second regression problem, called *distance*, requires the Euclidean distance between two points (vectors) $\vec{x}$ and $\vec{y}$ in n-dimensional space to be computed by the genetic programs (see Equation 5.5). The higher the dimension is chosen ($n = 3$ here) the more difficult the problem becomes.

The third problem is the well-known *spiral* classification [51] where two interlaced spirals have to be distinguished in two-dimensional data space. All data points of a spiral belong to the same class as visualized in Figure 5.3.

Finally, the *three chains* problem concatenates three rings of points that each represent a different data class (see Figure 5.4). Actually, one "ring" denotes a circle of 100 points in three-dimensional space whose positions are slightly noisy. The rings approach each other at five regions without leading to intersection. These regions determine the problem difficulty that may easily be scaled up or down depending on both the angle of the rings to one another and on the number of rings.

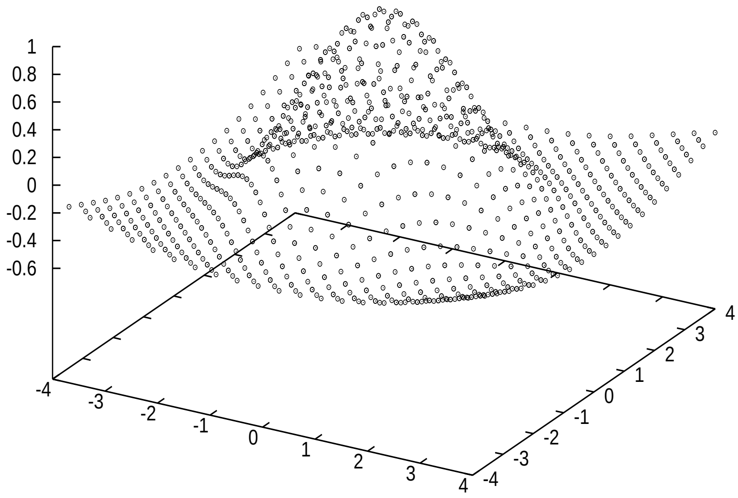

Figure 5.2: *mexican hat* problem.

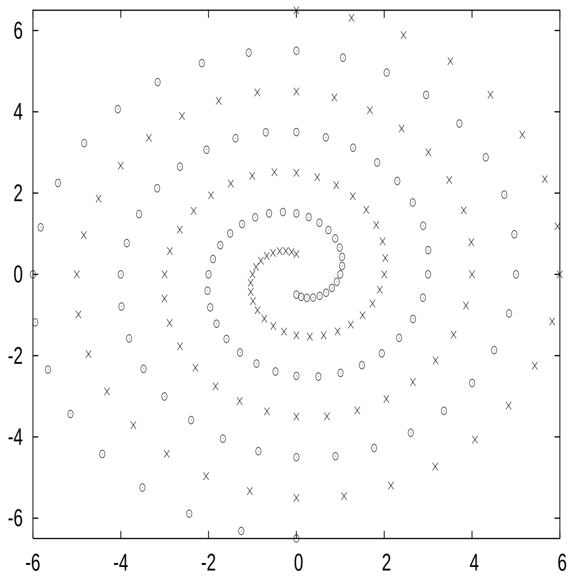

Figure 5.3: *spiral* problem.

5.8.2 Parameter Settings

Tables 5.1 and 5.2 summarize attributes of the data set that have been created for each problem. These include the input dimension, the output dimension, the ranges of input and output values as well as the number of training examples (fitness cases). Furthermore, problem-specific configurations of the LGP system are given that comprise the fitness function, the compositions of the function set, and the numbers of registers and constants.

It is important for the performance of linear GP to provide enough registers for calculation, especially if the input dimension is low. Therefore, the number of (calculation) registers – additional to the minimum number of registers that is required for the input data – is an important parameter (see also Section 6.1). In general, the number of registers determines the number of program paths that can be calculated in parallel. If it is not sufficient there

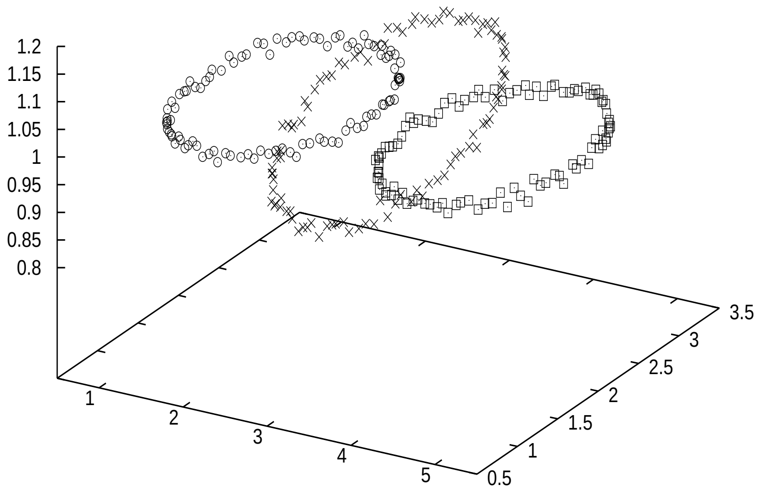

Figure 5.4: *three chains* problem.

may be too many conflicts by overwriting register contents within programs.

For the classifications tasks specified in Table 5.2 the fitness function is discrete. Fitness equals the *classification error* (CE) here, i.e., the number of wrongly classified inputs. For the approximation problems (see Table 5.1) the fitness is the continuous *sum of square output errors* (SSE).

The *spiral* problem applies an *interval classification* method, i.e., if the output is smaller than 0.5 it is interpreted as class 0, otherwise it is class 1. For the *three chains* problem we use an *error classification* method, instead. That is, the distance between the problem output and one of the given identifiers for the output classes (0, 1, or 2) must be smaller than 0.5 to be accepted as correct.

The instruction set used here for the *mexican hat* problem is incomplete, i.e., not sufficiently powerful to build the optimum solution. In particular, the exponential function e^x was not explicitly included. Since the basis constant e is an irrational number it may only be approximated by a finite number of program instruction. Multiple instances of an instruction in the instruction set, as used for the *distance* problem, increases its probability to be selected during initialization and mutation. In this way, the distribution of operator symbols within the population may be manipulated explicitly and is not only determined by the fitness selection. Only the instruction sets used for the classification problems include branches. Without branches these two problems cannot be solved completely.

General configurations of our linear GP system are given in Table 5.3. If not differently specified these configurations are used in all experiments. As already mentioned, always two tournament winners are either recombined or both of the two parents undergo mutation in the applied evolutionary Algoritm 2.1. Tournament selection is applied with a minimum of two participants per tournament. The tournament winners (parents) replace tournament losers for a (reproduction) probability of 100 percent.

In most experiments of this chapter macro operators are applied for a probability of 75 percent. On the one hand, this guarantees that the compared operators dominate the variation process. On the other hand, variation inside instructions is not reduced to zero but is still maintained by 25 percent micro mutations.

For all four test problems a maximum number of 200 instructions has proven to be sufficiently large to represent the optimum solution provided that the function set is complete,

Problem	*mexican hat*	*distance*
Problem type	Regression	Regression
Number of inputs	2	6
Number of outputs	1	1
Input range	$[-4.0, 4.0]$	$[0, 1]$
Output range	$[-1, 1]$	$[0, 1]$
Number of registers	$2 + 4$	$6 + 6$
Number of fitness cases	400	300
Fitness function	SSE	SSE
Instruction set	$\{+, -, \times, /, x^y\}$	$\{+, +, -, -, \times, \times, /, \sqrt{x}, x^2\}$
Set of constants	$\{1, .., 9\}$	$\{1, .., 9\}$

Table 5.1: Problem-specific parameter settings (regression problems).

Problem	*spiral*	*three chains*
Problem type	Classification	Classification
Number of inputs	2	3
Number of outputs	1	1
Number of output classes	2	3
Input range	$[-2\pi, 2\pi]$	$[0, 5]$
Output range	$\{0, 1\}$	$\{0, 1, 2\}$
Number of registers	$2 + 4$	$3 + 3$
Number of fitness cases	194	300
Fitness function	CE	CE
Instruction set	$\{+, -, \times, /, sin, cos, if >\}$	$\{+, -, \times, /, x^y, if >\}$
Set of constants	$\{1, .., 9\}$	$\{1, .., 9\}$

Table 5.2: Problem-specific parameter settings (classification problems).

Parameter	Setting
Number of runs	100
Number of generations	1000
Population size	1000
Tournament size	2
Maximum program length	200
Initial program length	5–15
Macro variations	75%
Micro mutations	25%
Reproduction	100%

Table 5.3: General parameter settings.

i.e., powerful enough. Actually, this maximum complexity bound allows similar (effective) program sizes to develop with most macro operators during 1000 generations – including segment variations and instruction variations. This in turn makes a comparison of prediction performance more fair in terms of the solution size.

If more insertions than deletions of code happen on average this tendency is referred to as an *explicit grow bias* of the variation operator. Table 5.4 gives an overview over the different configurations of insertion rates that are applied in the following experiments.

Bias Config.	B–1	B0	B1	Bmax
Insertions (%)	33	50	67	100
Deletions (%)	67	50	33	0
Ratio	1:2	1:1	2:1	1:0

Table 5.4: Different probabilities for insertions and deletions (macro operators). Configuration B1 induces an explicit grow bias by allowing two times more insertions than deletions. B–1 denotes a shrink bias, accordingly. Maximum growth tendency with Bmax. Configuration B0 is bias-free.

5.9 Experiments I: Segment Variations

All variation schemes that have been discussed above for the linear program representation involve single contiguous instruction *segments*. This section documents all experiments that have been conducted with these segment operators, which include recombination and segment mutation.

5.9.1 Comparison of Recombination Operators

In Tables 5.5 and 5.6 the different approaches to recombination operators are compared in terms of their influence on the prediction performance, code growth and the probability distribution of variation effects. The *mean* prediction error is calculated over 100 independent runs together with the statistical standard error (*std.*). The number of hits, i.e., the number of successful runs, is not given here because the optimum has almost never been found by any crossover operator during a period of 1000 generations. This is true for both benchmark problems applied here, *spiral* and *mexican hat*. As described in Section 5.8.1 the two problems are structured quite differently and belong to different problem classes, i.e., classification and approximation. To reduce noise through unequal initial populations, each test series is performed with the same set of 100 random seeds.

The program length is averaged over all programs that are created during a run *and* over the 100 trails. Thus, the average effective program length gives more precise information about the average calculation time that is necessary for executing a program during a run. Recall that the effective length corresponds to the number of executed instructions in our system (see Chapter 3.2.1). The proportion of effective code p_{eff} is given in percent with $p_{noneff} = 100 - p_{eff}$ calculates the rate of (structural) introns.

The absolute length l_{abs} includes all instructions while the effective length l_{eff} counts instructions that are effective. As indicated in Section 5.7 the ratio of effective length and absolute length $\frac{l_{eff}}{l_{abs}}$ is an important parameter when using linear crossover. It determines the average number of effective instructions that may be deleted or selected from a parent program. This, in turn, influences the average effective step size as defined in Section 5.3.

Additionally, Tables 5.5 and 5.6 show the average proportion of constructive, neutral and noneffective variation effects among all variations during a run. The rates of destructive and effective variations are obvious then.

Operator	Config.	SSE		Length			Variations (%)		
		mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
cross		15.4	1.5	180	67	37	4.9	26	22
	effinit	13.3	1.4	178	65	37	5.0	26	22
	effdel	14.3	1.4	171	68	34	5.9	22	18
onepoint		21.9	1.3	188	66	35	2.8	78	69
oneseg		12.1	1.3	158	57	36	4.5	27	24
effcross		26.9	2.5	51	51	100	6.6	32	9
	effinit	6.1	0.8	111	111	100	9.4	12	1.8

Table 5.5: *mexican hat*: Comparison of different recombination operators and configurations. Average results over 100 runs after 1000 generations.

Operator	Config.	CE		Length			Variations (%)		
		mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
cross		26.1	0.7	185	102	55	3.6	23	14
	effinit	24.3	0.7	183	104	57	3.5	24	14
	effdel	25.2	0.7	184	95	51	4.5	20	12
onepoint		32.0	0.9	190	89	47	0.9	81	32
oneseg		24.0	0.8	164	85	52	2.5	26	18
effcross		26.0	0.7	162	162	100	4.0	22	2.4
	effinit	18.8	0.7	164	164	100	3.9	20	0.6

Table 5.6: *spiral*: Comparison of different recombination operators and configurations. Average results over 100 runs after 1000 generations.

Two-point crossover (cross) performs better than one-point crossover (onepoint). Interestingly, even if the average (absolute) step size is larger with only one crossover point per individual, a much higher proportion of operations is neutral. In case of the *mexican hat* problem most of these variations are noneffective, too, i.e., do not alter the effective solution. Since the endpoints of segments are always the same an exchange of (effectively) identical segments becomes much more likely.

Only slightly better results are obtained with one-segment recombination (oneseg) compared to standard crossover. We argued in Section 5.7.3 that those may reduce the variation step size. However, since the program size grows similarly large on average and because segment length is unrestricted with both variants this effect may be hardly relevant here.

The effective crossover variant effcross is implemented in such a way that the (structural) noneffective code is removed completely after each variation (see Section 5.7.1). In doing so, the deletion of instruction segments as well as all micro mutations automatically becomes effective. Even if this is not necessarily valid for a segment insertion, too, the whole exchange of code is mostly effective here. The main reason why the prediction performance may become worse is a higher effective crossover step size due to the lack of noneffective instructions. This makes a stepwise improvement of solutions more difficult since the average amount of change may be expected significantly higher for the effective code (see also Section 5.7.4). Another reason might result from the fact that longer and more specific solutions (effective code) are more brittle during variation.

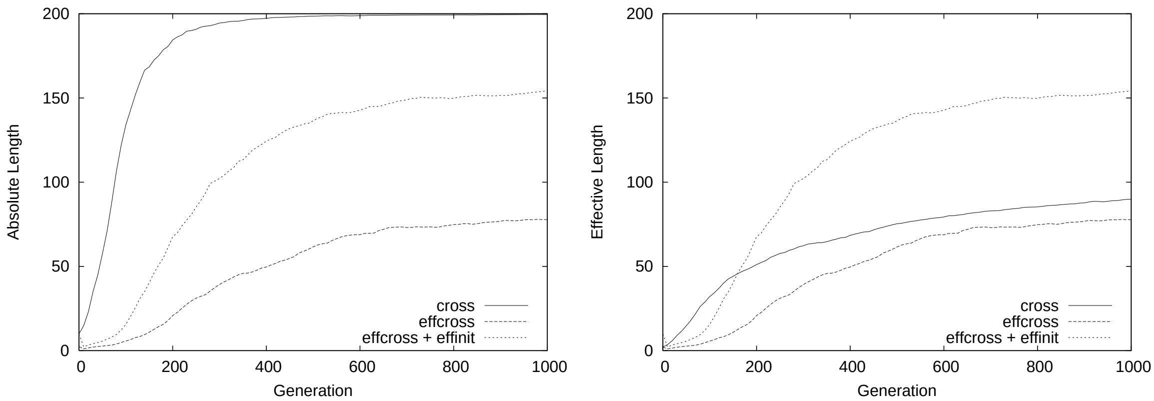

Figure 5.5: *mexican hat*: Development of absolute program length (left) and effective program length (right) for different crossover operators. Code growth significantly reduced by removing the noneffective code. Average figures over 100 runs.

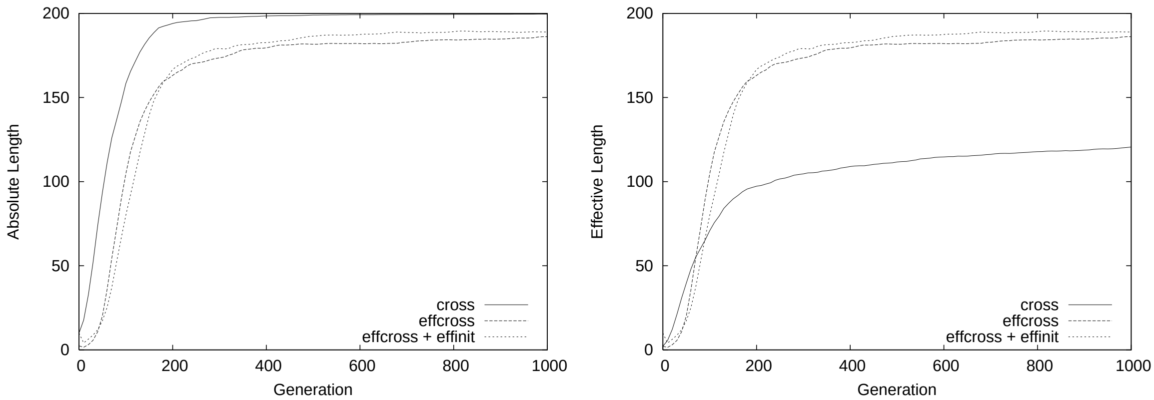

Figure 5.6: *spiral*: Development of absolute program length (left) and effective program length (right) for different crossover operators. Removal of structural introns compensated by more semantic introns. Note that absolute length and effective length are the same with effcross. Average figures over 100 runs.

The continuous loss of (structurally) noneffective code is compensated by a larger effective code only if the problem definition allows a sufficient amount of semantic introns (as part of the effective code, see Section 3.2). Recall that the ability to create semantic introns depends on the configuration of the instruction set. On the other hand, a sufficient replacement depends on the question how far the solution finding for a problem profits from a growth of effective code. In contrast to the *mexican hat* problem, the discrete *spiral* problem allows good solutions to incorporate a relatively large amount of effective code. This is facilitated by using branching instructions that offer an additional potential for creating semantic intron code.

Figures 5.5 and 5.6 compare the development of average lengths and average effective lengths in the population for both test problems. We just note here that the length of best solutions develops almost identically to the average length if averaged over multiple runs. The standard deviation of effective lengths in the population is smaller than 10 instructions on average (not specified in Tables 5.5 and 5.6). One reason for the small standard deviation is the early restriction of (absolute) code growth for this genetic operator by the maximum size limit. The standard deviation of absolute lengths is even smaller and converges to 0 if the average length converges to the maximum length. As one can see program growth is significantly reduced for *mexican hat* in effcross runs. Actually, absolute programs do not even become as long here as the effective code in cross runs. For the *spiral* classification, instead, the permanent loss of noneffective code is much better compensated by semantic intron code. The average program size nearly reaches the maximum length just like in runs with normal crossover.

The *mexican hat* results demonstrate here that the existence of structurally noneffective code in linear GP offers an advantage over semantic introns because the former may be created more easily by evolution and independently from the function set. In other words, the emergence of semantic intron code is more suppressed in the presence of a structural introns. By doing so, the (structurally) noneffective code *reduces* the size of effective programs (*implicit parsimony pressure*, see also Section 7.4.1).

Furthermore, Figure 5.5 reveals that the removal of noneffective code is especially destructive at the beginning of an effcross run where effective solutions are most brittle since they have not developed a sufficient (effective) size for compensation yet. Programs become so small after the first generation that many are structurally identical – and even more are semantically identical. That is, the initial loss of code is accompanied by a high loss of diversity in the population. Hence, it is not surprising that the effective crossover variant profits much more from an effective initialization (effinit, see also Section 6.6) in terms of the prediction quality than this is found with normal crossover. Effective initialization means that the initial programs are created completely effectively while the absolute amount of genetic material stays the same. Due to this special form of initialization the program size doubles in Figure 5.5 probably because semantic introns may be created sufficiently then. With the *spiral* problem, by comparison, the initial phase of code loss occurs to be much shorter (see Figure 5.5).

There is still a small proportion of noneffective operations that occurs with the effcross variant in Tables 5.5 and 5.6. This may result from the exchange of segments which are (effectively) identical. Such a situation becomes particularly likely if programs and segments, accordingly, only comprise a few (effective) instructions or if many programs are identical at the beginning of a run.

Only slightly better results have been found compared to the standard approach if it is only cared for that crossover operations are effective, i.e., delete at least one effective instruction (effdel). Because the rate of noneffective variations is not reduced significantly and because of the large absolute step size of crossover, we may assume that most variations are already effective when using standard crossover.

In general, the different crossover operators and configurations performed more similar than what might have been expected. One reason is the maximum segment length (and thus the maximum step size) that is restricted by the program size only. Programs, however, grow similarly large with almost all recombination operators what is only partly a result of the complexity bound (see Chapter 9).

5.9.2 Comparison with Segment Mutations

Tables 5.7 and 5.8 list the results that have been obtained with segment mutations. Recall that variant segmut replaces an instruction segment by a random segment of arbitrary length while the onesegmut variant deletes segments and inserts random ones in separate genetic operations. From a technical point of view, the first variant operates similar to standard crossover (cross) while the latter variant corresponds to one-segment recombination (oneseg).

All compared segment operators are *unbiased* in terms of the program length, i.e., do not promote code growth explicitly. Without fitness pressure (flat fitness landscape) there would be no relevant increase of program length. Therefore and for the purpose of a fair comparison with recombination, segment mutations have been implemented in Section 5.7.5 such that the maximum segment length of both insertions and deletions depends on the length of programs in the population. This may, however, guarantee similar segment lengths and step sizes as recombination only if programs grow similarly large.

Operator	SSE		Length			Variations (%)		
	mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
segmut	12.6	1.3	72	28	39	5.1	26	18
effsegmut	4.1	0.3	31	23	76	7.6	19	6
onesegmut	4.2	0.5	92	38	42	4.6	26	21
effonesegmut	2.0	0.1	43	32	74	7.3	19	8

Table 5.7: *mexican hat*: Comparison of different segment mutation operators. Average results over 100 runs after 1000 generations.

Operator	CE		Length			Variations (%)		
	mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
segmut	27.3	0.7	121	61	50	3.3	25	15
effsegmut	28.1	0.7	35	29	82	5.3	18	4
onesegmut	21.2	0.6	126	65	51	2.4	27	19
effonesegmut	19.1	0.5	67	54	81	4.1	18	4

Table 5.8: *spiral*: Comparison of different segment mutation operators. Average results over 100 runs after 1000 generations.

It is an important result that recombination does not perform better than segment mutations here. Recall from the discussion in Section 5.7.7 that this may be taken as an argument against the building block hypothesis. Interestingly, with two-segment mutations (segmut) the prediction performance is hardly different from crossover. Only one-segment mutations (onesegmut) show more significant improvements compared to one-segment recombination, especially for the *mexican hat* problem. As noted above, *mexican hat* is better solved with a more reduced growth of programs, in contrast to the *spiral* problem.

A better performance of one-segment mutations (compared to two-segment mutations) may not only result from (a reduction of the absolute step size by) a smaller program size which is almost equally reduced for both types of segment mutations here. Instead, the twice as large absolute step size of two-segment variations (according to Definition 5.3) must be responsible for this. Beyond a certain average step size results may be only slightly different. We will demonstrate in Section 9.8.6 that larger segments are correlated to larger fitness changes only until a certain segment length. This is argued to be a result of the imperative program structure and its more-or-less linear data flow (see Section 3.3).

Besides, one-segment and two-segment mutations differ more strongly in the prediction error than the two corresponding recombination operators in Section 5.9.1. Note that the average step size of mutations is smaller already because of a smaller size of solutions.

It is an interesting question why smaller (effective) programs occur with segment mutations than with recombination although in both cases the segment size is limited by the program size only. Possible reasons for this will be discussed in Section 9.9.1. We only note here that the difference in program size increases with a larger maximum program length (200 instructions here) since recombination is much more influenced by this.

A slightly better performance but definitely smaller solutions are obtained if it is explicitly guaranteed that each instruction of an *inserted* segment is created effectively (effonesegmut). On the one hand, this reduces the rate of noneffective (neutral) variations. Noneffective variations still occur here for a small probability mostly because of the 25 percent free micro mutations that are applied together with each macro operator. Only some noneffective operations may result from segment deletions, too. Note that it is not explicitly guaranteed here that a deletion is effective.

On the other hand, the proportion of (structurally) noneffective instructions is significantly smaller compared to using free segment mutations. First, such noneffective instructions are not directly created, but may occur only indirectly by deactivations of depending instructions. Second, deleted segments may still contain noneffective instructions while inserted segments are fully effective (see next section). This corresponds to an explicit shrink bias in terms of the noneffective code. Hence, the effective step size may hardly be reduced by a higher rate of structural introns in programs.

Exchanging fully effective segments does not seem to have a negative influence on the prediction performance here in terms of a higher effective step size. This may be compensated at least partly by smaller absolute step sizes that result from the smaller programs.

When using effective two-segment mutations (effsegmut) code growth is even more reduced than this occurs with effective one-segment mutations. First, this operator allows noneffective code to be *replaced* by effective code but not vice versa. Second, the standard deviation of segment lengths over a run is smaller than this occurs with the more probabilistic one-segment mutations (effonesegmut). Both factors have a negative influence on code growth and may become relevant here since most positive influences are excluded as far as possible. As for the *spiral* problem, code growth may be too much restricted to let more efficient solutions emerge. Instead, the performance is improved significantly with the *mexican hat* problem.

But why is the program length not increased by semantic introns here as this has been observed with effective crossover (effcross) above ? Apparently, the creation of both semantic and structural introns is much more limited when using (effective) segment mutations (see Section 9.9.1).

5.9.3 Crossover Rate

In Section 5.8 we have used a configuration of variation rates that assigns 75 percent to macro variations and 25 percent to micro mutations. In this way, it is guaranteed that the macro operator dominates variation while still enough modifications happen inside instructions (by micro mutations).

Tables 5.9 and 5.10 compare results for different crossover rates p_{cross} in percent while the probability for micro mutations is $p_{micromut} = 100 - p_{cross}$, accordingly. Only one variation is applied at a time, i.e., between two fitness evaluations. The more micro mutations are applied, the smaller the average step size becomes, but the more variations become noneffective and neutral, too. The advantage of smaller step sizes seems to outweigh the disadvantage of less effective variations here. In both problem cases, the best performance has been achieved with the smallest crossover rates (10 percent here). Although only a few macro variations are responsible for code growth then, programs still grow almost equally large which is a result of the unrestricted step size of crossover.

Crossover (%)	SSE		Length			Variations (%)		
	mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
10	**9.0**	1.2	121	54	45	1.5	46	44
25	12.7	1.5	150	64	43	1.8	42	40
50	13.8	1.4	170	64	38	2.7	36	33
75	15.4	1.5	180	67	37	4.9	26	22
100	23.5	1.4	182	48	26	6.1	27	22

Table 5.9: *mexican hat*: Comparison of different crossover rates (in percent). Average results over 100 runs after 1000 generations.

Crossover (%)	CE		Length			Variations (%)		
	mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
10	**14.9**	0.7	142	88	62	0.6	42	30
25	17.6	0.7	164	99	60	0.9	39	27
50	23.0	0.7	178	99	56	1.9	31	22
75	26.1	0.7	185	102	55	3.6	23	14
100	34.5	0.6	187	98	53	5.8	17	8

Table 5.10: *spiral*: Comparison of different crossover rates (in percent). Average results over 100 runs after 1000 generations.

There is an especially large decrease in performance if crossover is applied exclusively compared to using micro mutations for 25 percent. Crossover may only recombine program components (instructions) that already exist in the previous generation but does not introduce new instructions. By the influence of selection and reproduction the concentration of certain instructions may be reduced significantly. This is avoided by applying mutations at least for a small percentage.

5.9.4 Analysis of Crossover Parameters

Linear crossover has been defined as the mutual exchange of a contiguous sequence of instructions between two individual programs in Section 5.7.1. In the following the influence of the three crossover parameters

- ☐ Maximum length of segment
- ☐ Maximum difference in segment length
- ☐ Maximum distance of crossover points

on prediction performance, program growth and, variation effects is analysed. Note that the term *crossover point* refers to the first absolute position of a segment. All lengths and distances are measured in instructions and are selected uniformly distributed from the predefined maximum ranges.

Tables 5.11 and 5.12 show the results of different maximum thresholds for the segment length, ranging from two[2] instructions only to all instructions of a program which does

[2]Code growth would not be possible with maximum segment length 1 since crossover exchanges, by definition, at least one instruction.

not impose any restrictions. Segment lengths are selected uniformly distributed from the maximum range. For both problems, *mexican hat* and *spiral*, the best fitness has been found if at most 5 instructions are allowed to be exchanged. Especially in case of the *spiral* problem the growth of programs seems to be too restricted with segment length 2 to develop competitive solutions.

Maximum	SSE		Length			Variations (%)		
Segment Length	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
2	4.3	0.6	50	31	63	3.8	29	26
5	**3.5**	0.5	107	50	47	3.5	31	28
10	8.5	1.2	146	58	40	3.6	31	28
20	10.9	1.3	169	65	38	3.9	30	26
50	13.3	1.3	177	65	37	4.5	27	24
–	15.4	1.5	180	67	37	4.9	26	22

Table 5.11: *mexican hat*: Effect of maximum segment length using crossover (cross). Average results over 100 runs after 1000 generations.

Maximum	CE		Length			Variations (%)		
Segment Length	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
2	17.4	0.6	54	38	70	1.6	29	21
5	**12.8**	0.6	125	77	61	1.7	33	20
10	18.8	0.6	166	99	60	2.0	29	18
20	22.0	0.7	180	102	56	2.7	26	17
50	24.8	0.7	185	103	56	3.2	24	15
—	26.1	0.7	185	102	55	3.6	23	14

Table 5.12: *spiral*: Effect of maximum segment length using crossover (cross). Average results over 100 runs after 1000 generations.

Basically, the relative influence on the average fitness decreases with larger maximum segment lengths here because of the following reasons. First, the average segment length is relatively small even for unrestricted two-point crossover (less than 25 percent of the program length on average). Second, because of a more-or-less linear data flow the influence of the segment length may be proportional to the program length only to a certain degree (see also Section 9.8.6). Finally, code growth is reduced significantly only when using relatively small upper bounds for the segment length. Due to restrictions by the maximum program length (200 instructions here) there is no significant difference in the average program length beyond a certain maximum segment length anymore. A reduction of program lengths indirectly influences the average segment length again since a segment may not be larger than the program from which it originates.

The rate of effective code decreases with the maximum segment length, i.e., the rate of noneffective code increases. Since smaller segments mean smaller (absolute) step sizes there is less need to reduce the effective step size of crossover by developing more intron code (see also Chapter 9). It is interesting to note that the rates of noneffective and neutral variations are less affected in Tables 5.11 and 5.12 by a restriction of the segment length. The higher probability of smaller replacements to be noneffective or effectively identical is mostly compensated here by a higher proportion of effective code.

These results imply that the average variation step size of (unrestricted) standard crossover is too large. A strong restriction of the segment length, however, may not be regarded as real crossover anymore. At least, the idea of combining advantageous building blocks from different programs may be questioned if the building blocks only comprise a few (effective) instructions. This might be used as another argument against the building block hypothesis (see Section 5.7.7).

Max. Segment	SSE		Length			Variations (%)		
Length Difference	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
1	**3.6**	0.5	48	29	60	5.4	24	21
2	4.4	0.7	77	41	54	5.2	25	22
5	7.7	1.1	124	56	45	5.2	24	21
10	10.1	1.2	159	61	39	5.0	25	22
20	13.7	1.4	175	65	37	4.9	25	22
50	15.4	1.4	183	66	36	4.9	26	23
–	15.4	1.5	180	67	37	4.9	26	22

Table 5.13: *mexican hat*: Effect of maximum difference in segment length using crossover (cross). Average results over 100 runs after 1000 generations.

Max. Segment	CE		Length			Variations (%)		
Length Difference	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
1	20.8	0.6	56	41	73	3.6	22	14
2	**18.5**	0.7	91	63	69	3.6	23	13
5	20.6	0.7	151	91	60	3.4	25	15
10	23.3	0.7	173	97	56	3.6	24	15
20	24.6	0.6	182	100	55	3.5	24	15
50	25.5	0.6	186	101	55	3.6	23	15
—	26.1	0.7	185	102	55	3.6	23	14

Table 5.14: *spiral*: Effect of maximum difference in segment length using crossover (cross). Average results over 100 runs after 1000 generations.

For the following considerations we assume that the segment length is unrestricted again. Instead, we limit the maximum *difference* in length between the two exchanged crossover segments. For this purpose, we select one segment freely in one of the parents. The position of the second segment is selected without restrictions from the other parent. Only for the length of this segment it is guaranteed that a maximum distance from the length of the first segment is not exceeded. In this way, a form of *size fair crossover* is implemented in linear GP (see also Section 5.7.1). Langdon found that size fair crossover reduces bloat in (tree-based) genetic programming [56].

In general, Tables 5.13 and 5.14 document similar results as found with a restriction of the segment length above. This may be interpreted in such a way that a smaller maximum difference in segment length reduces the crossover step size in a similar way as this results from using a smaller maximum segment length. The more similar the lengths of the exchanged segments are the less programs can increase in length during a crossover operation.

Conclusively, the potential speed of code growth depends on both the size and the difference in size of the exchanged code fragments. However, while an exchange of very small segments may hardly be regarded as crossover, this is not the case for the size fair implementation. On the contrary, size fair crossover is even more closely related to crossover in nature. Crossed DNS strings are not only of a similar length but happen at similar positions (crossover points), too. The distance of crossover points is investigated in the next experiment.

Maximum	SSE		Length			Variations (%)		
Point Distance	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
0	25.1	1.3	184	60	33	1.5	82	75
2	21.3	1.4	182	79	43	3.3	50	45
5	20.2	1.4	181	77	43	3.8	41	37
10	19.4	1.5	181	80	44	4.5	33	30
20	18.5	1.5	180	75	42	4.4	31	29
50	17.1	1.4	180	71	40	4.4	29	27
–	**15.4**	1.5	180	67	37	4.9	26	22

Table 5.15: *mexican hat*: Effect of maximum distance of crossover points (cross). Average results over 100 runs after 1000 generations.

Maximum	CE		Length			Variations (%)		
Point Distance	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
0	26.7	0.7	186	90	49	0.5	82	47
2	22.6	0.8	183	87	47	1.6	52	30
5	21.5	0.6	182	98	54	2.0	41	24
10	**20.3**	0.6	182	98	54	2.2	36	22
20	22.5	0.7	181	100	55	2.6	32	20
50	25.7	0.6	185	103	55	2.9	28	18
—	26.1	0.7	185	102	55	3.6	23	14

Table 5.16: *spiral*: Effect of maximum distance of crossover points (cross). Average results over 100 runs after 1000 generations.

Different maximum distances of crossover points in the two parent individuals are tested in Tables 5.15 and 5.16. In contrast to the results that have been found with the other crossover parameters, the results are more different here for both test problems. While *mexican hat* is clearly better solved without such a restriction of variation freedom, the *spiral* problem seems to profit slightly from more similar positions of crossover points. If the crossover points are selected below a certain optimum distance, however, the prediction error increases again. This is especially true for minimum distance 0. Apparently, if only equal crossover points are allowed evolution is restricted significantly in its ability to move code fragments from one program region to another. This may lead to a loss of code diversity among the population individuals. We may conclude that a free choice of crossover points in *both* parents is important, at least to a certain extent.

In comparison with the two other parameters the maximum distance of crossover points has a lower impact on the (effective) program size. Instead, the rate of noneffective (and thus neutral) variations increases significantly if the crossover points are chosen more similarly, especially with distance 0. This is a direct hint that the diversity of effective code is negatively affected here because (effectively) identical segments are exchanged for a higher probability. Similar observations have been made with one-point crossover in Section 5.9.1 where the endpoints of segments – instead of the starting points here – are always identical.

Consequently, only if both smaller differences in segment lengths and smaller distances of crossover points have a positive influence on the performance, homologous crossover – combining both attributes – may be beneficial (see Section 5.7.1). Otherwise, these two criteria may work against each other.

5.9.5 Explicit Introns

Many implicit introns in linear genetic programs reduce the number of effective instructions that may be exchanged by crossover. However, this positive influence on the effective crossover step size is limited by reactivations of intron instructions. The higher the intron rate is the more of such side-effects may occur, on average. We test whether explicitly defined introns (EDIs, see Section 5.7.6) may provide a more reliable reduction of effective step sizes.

On the other hand, explicit introns constitute a method for controlling the number of coding (non-EDI) instructions, i.e, the actual program complexity. Since both implicit (structural) introns and explicit introns can be removed efficiently before the fitness calculation in linear GP (see Section 3.2.1) an acceleration of runtime may only result from a smaller effective size.

We have seen above that the growth of effective code is accelerated significantly with crossover if all noneffective instructions are removed directly after each operation. From this we followed that without structural introns there is more need for expanding the effective code by semantic introns. While such effective variations necessarily increase the effective step size explicit introns have been introduced for exactly the opposite reason. We may assume that the creation of semantic introns is more suppressed in the presence of explicit introns than this is already true in the presence of structural introns.

In both Tables 5.17 and 5.18 a maximum initialization with explicit introns reduces the average size of effective code almost by half and produces the best prediction results. Implicit introns emerge less, depending on the amount of empty instructions that is provided in the initial population. Note that in all configurations the same amount of non-empty initial instructions is used (10 instructions on average). It may be noted also that explicit introns are not allowed to follow directly after a branch instruction in programs. This has been found to reduce the probability significantly that a branch is followed by an (effective) operation and, therefore, produces worse results.

Even though the rate of effective instructions decreases almost by half if the initial population is filled up with explicit introns, intron segments are not exchanged more frequently. This is why the rate of noneffective and neutral operations stays more-or-less the same in Tables 5.17 and 5.18. In the first place, this is a result of the large unrestricted step size of crossover.

In general, the larger the initial programs are the more quickly the average program size grows up to the maximum (see Figure 5.7). This is simply due to the absolute step size of unrestricted crossover that increases proportionally with the absolute program size. As long as programs grow, the step size grows, too. Only after code growth has been stopped by the maximum length bound or if the size of initial programs is already maximum, the average absolute step size is constant.

Initial EDIs	SSE		Length			EDIs		Variations (%)		
n×	*mean*	*std.*	*abs.*	*eff.*	%	#	%	*constr.*	*neutral*	*noneff.*
0×	15.4	1.4	180	67	37	—	—	4.9	26	22
1×	11.4	1.3	186	50	27	73	39	4.9	26	22
2×	8.5	1.1	190	42	22	102	54	4.8	26	23
4×	7.5	1.1	194	37	19	123	63	4.8	26	23
max	**5.6**	0.7	200	30	15	147	74	4.5	28	25

Table 5.17: *mexican hat*: Effect of empty instructions (EDIs) on crossover (cross). Number of empty instructions in an initial program equals n times the number of non-empty instructions (10 on average). Average results over 100 runs after 1000 generations.

Initial EDIs	CE		Length			EDIs		Variations (%)		
n×	*mean*	*std.*	*abs.*	*eff.*	%	#	%	*constr.*	*neutral*	*noneff.*
0×	26.1	0.7	185	102	55	—	—	3.6	23	14
1×	25.4	0.7	190	75	40	57	30	3.4	23	16
2×	24.2	0.7	193	67	35	84	44	3.3	23	15
4×	22.2	0.7	195	59	30	100	51	3.3	24	18
max	**18.1**	0.6	200	54	27	121	61	2.7	24	16

Table 5.18: *spiral*: Effect of empty instructions (EDIs) on crossover (cross). Number of empty instructions in an initial program equals n times the number of non-empty instructions (10 on average). Average results over 100 runs after 1000 generations.

If empty instructions are seeded additionally into the initial population the effective step size decreases for two reasons. First, the more explicit introns are provided initially the less implicit (structural) introns are found and the smaller is the proportion of effective code (due to less semantic introns). Because explicit introns are independent from the structural and semantic program context they allow the size of effective code to be more independent from the absolute program size. Second, the effective step size may not be increased indirectly by reactivations of introns, if these comprise empty instructions.

Figure 5.7 illustrates the development of average program lengths and average intron rates in the population for different initial amounts of explicit introns. Without using explicit introns the implicit (structural) introns grow quickly at the beginning of a run until the program length is almost maximum. After that point in about generation 200 the noneffective code decreases slowly towards the end of a run due to a still growing effective code, which replaces noneffective instructions more and more by effective ones.

If explicit introns are provided the proportion of implicit introns develops smaller. If the initial programs are completely filled up with explicit introns, the implicit intron rate reaches only about 10 percent of the maximum length at the end of runs with both test problems.

Besides, the more explicit introns are provided in initial programs the smaller the effective code develops. On the one hand, like structural introns such introns take away pressure from the effective code to grow and to develop semantic introns (see Section 5.9.1). Recall that semantic introns are usually more difficult to create, depending on the problem configuration. On the other hand, this may not be achieved just by using longer initial programs. Because of context dependencies longer programs do not only imply a higher amount of implicitly noneffective code but usually more effective code, too.

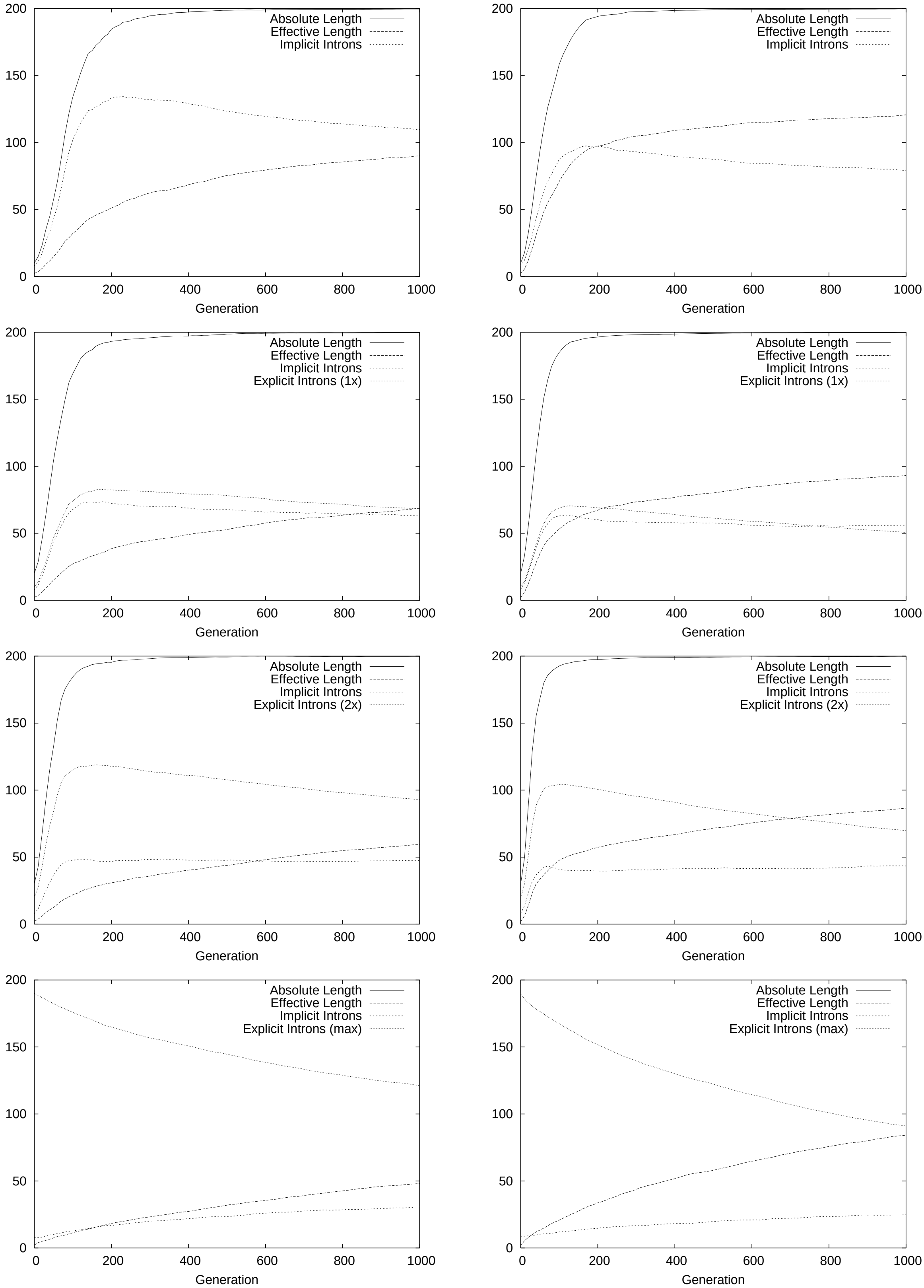

Figure 5.7: *mexican hat* (left), *spiral* (right): Development of program lengths and intron rates over the generations and for different initial amounts of explicit introns.

The explicit introns hardly affect the final effective program size at the end of run. Note that the effective size is more strongly determined by a programs' ability to solve a certain problem task, i.e., by its fitness. Nevertheless, the effective code grows more slowly (linear) over the generations.

At the beginning explicit introns spread fast within the population. This depends strongly on the amount of such empty instructions in the initial population. However, the implicit (structural) introns grow about as fast as the explicit ones if their initial numbers are the same (see Figure 5.7). Then both types coexist for certain quantities during the whole run. Recall that already structural introns emerge easily in linear GP. Hence, explicit introns do *not* displace implicit (structural) introns in the course of a run. It is important to provide a high amount of explicit introns right from the start.

5.10 Instruction Mutations

The experimental results from Section 5.9 have confirmed two important presumptions. On the one hand, when using recombination best results were obtained with a relatively small segment length. On the other hand, segment recombination has not been found to be more powerful than segment mutation. Both aspects motivate the use of macro mutations that insert or delete a single (full) instruction (*instruction mutations*) only. Different mutation operators and variation techniques of that kind will be described in this section. First, the following considerations will point out why especially linear programs are probably better developed by using minimum mutations exclusively.

5.10.1 Minimum Mutation Step Size

Why small variation steps may promise better results in genetic programming ? As noted above, small variation steps allow a more precise approximation in general. This is due to the fact that small structural step sizes imply small semantic step sizes for a higher probability. Nevertheless, changing even smallest symbols in a genetic program may still induce relatively large semantic changes, on average (see Chapter 8). Therefore, a too strong deceleration of the global search progress by too small step sizes may be rather unlikely. This is in contrast to other evolutionary algorithms, like evolution strategies, that operate on a numerical representation in a more continuous search space. Also note that, theoretically, step sizes on real-valued parameter values may be arbitrarily small.

Using small variation steps in GP better corresponds to the biological pattern, too. Most mutations in nature affect only small parts of the genotype. This is also true for changes caused by crossover of DNA strands due to its perfect alignment (homologous crossover) and many identical genes. Otherwise, a high rate of viable offsprings would not be possible. In nature genotype variations are expressed in relatively small changes of the phenotype only. As noted in the last section, crossover in GP works quite differently. Most crossover operations have a high destructive influence on both the genotype representations and their phenotypes, i.e., the program behavior. One reason is that the selection of crossover points in both parents as well as the size and structure of the two exchanged subprograms are much less constrained. Another reason may be that the functionality of building blocks in programs (instructions) is less place bound than the builing blocks of DNA (genes).

The following arguments suggest a higher potential of mutations in linear GP than this is possible in tree-based GP. In particular, there are some basic reasons that let us favor minimum mutations step sizes on the (absolute) linear program structure. That means only one instruction may either be inserted or deleted on the macro level. On the level of micro code, i.e, inside instructions, minimum components of instructions are exchanged as usual.

First, already single micro mutations that exchange a register index in an instruction may change the data flow within a linear program heavily (see Section 3.3). Several instructions that precede the mutated instruction may become effective or noneffective respectively. Thus, the effective step size of instruction mutations (see Definition 5.4) may involve many instructions even if the absolute step size is minimum.

Second, the linear program representation can be manipulated with a *high degree of freedom.* Already by definition, its graph-structured data flow allows a higher variability than a tree due to multiple connections of nodes. This makes a constant realization of minimum macro variations possible at each position of the program. In a tree it is rather difficult to delete or insert a small group of nodes at an arbitrary position. Complete subtrees might be removed during such operations to satisfy the higher constraints of the tree structure.

In linear GP the depending substructures do not get lost when deleting or inserting an instruction but remain within the imperative representation as inactive code or as non-visited components of the data flow graph, respectively (see Section 3.3). The *existence of structural noneffective code* in linear genetic programs prevents a loss of genetic material. Code that has been deactivated in a program may already become active again after the next variation.

A tree structure is less suitable to be varied by small macro (subtree) mutations exclusively, since modification of upper program parts usually involve bigger parts of code. Smaller structural changes are only possible in trees if smaller subtrees are replaced that are located close to the leafs.

In contrast to that recombination may be criticized to be less suited for linear GP for the following reasons. In tree programs crossover and mutation points can be expected to have a stronger influence on program semantics the closer they are to the root (see Section 5.5). In a linear program each position of an instruction may have a more similar influence on program semantics. Recall that the underlying graph representation is restricted in width through the provided number of registers (see Section 3.3).

Another reason against using linear recombination is that usually the contents of many effective registers are changed simultaneously. The reason lies again in the rather narrow data flow graph of linear genetic programs. Such a graph is disrupted easily when applying crossover to the imperative program structure by what several program paths may be redirected simultaneously. As a result, crossover step sizes may become quite large, on average. In tree-based GP, by comparison, crossover only affects a single point in data flow that is the root of the exchanged subtree.

However, it has to be noted that to a certain degree the effective step size of linear crossover is decreased implicitly by increasing the proportion of structural introns (see Chapter 9). Inactive instructions may emerge at each position in a linear program for (almost) the same probability. In tree programs the creation of (necessarily semantic) introns is more limited, especially at higher node levels. Additionally, the effect of linear crossover may be reduced more directly than tree crossover by using a maximum size limit for the exchanged instruction segments.

As discussed in Section 5.7.7, various researchers investigated mutation operators for tree-based GP. O'Reilly and Oppacher [70] minimze the average amount of structural change as far as possible. Nonetheless, this may only be a compromise between a restriction of the variation freedom, on the one hand, and larger step sizes by loss of code, on the other hand (see also discussion in Section 7.5). Chellapilla [24] defines different types of mutation operators for tree programs ranging from the exchange of single nodes of the same arity (micro mutation) to the exchange of complete subtrees (macro mutation). His main interest, however, is not in a reduction of variation step sizes. Instead, he allows several operators to be applied successively to the same individual.

5.10.2 Macro Mutations

As noted above, we only regard instruction mutations in this section. Such macro mutations vary program length with a minimum effect on the program structure here by inserting and deleting a single instruction only. In other words, they induce a minimum step size on the instruction level. On the functional level a single instruction node is inserted in or deleted from the program graph, together with all its connecting edges.

We do not regard macro mutations that *exchange* an instruction or change the *position* of an existing program instruction only. Both variations are more destructive, i.e., imply a larger variation step size, since they include a deletion and an insertion at the same time. This is true even if, in the first case, deletion and insertion happen at the same program position and, in the second case, the inserted instruction originates from the same individual. Another important argument against substitutions of single instructions is that these do not vary the program length. If only single instructions would be exchanged there is no code growth possible at all. In general, substitutions may be explicitly length-biased only by applying larger absolute step sizes for either deletion or insertion.

Algorithm 5.4 has a similar structure as Algorithm 5.3. We will see below that an explicit grow bias ($p_{ins} > p_{del}$) may have a positive influence on the performance especially if only single effective instructions are added or removed.

Algorithm 5.4 (*(effective) instruction mutation*)
Parameters: insertion rate p_{ins}; deletion rate p_{del}; maximum program length l_{max}; minimum program length l_{min}.

1. Randomly select macro mutation type *insertion* | *deletion* for probability $p_{ins} \mid p_{del}$ and $p_{ins} + p_{del} = 1$.
2. Randomly select an instruction at a program position i (mutation point).
3. If $l(gp) < l_{max}$ and (*insertion* or $l(gp) = l_{min}$) then
   (a) Insert a random instruction at position i.
   (b) If *effective mutation* then
      i. If instruction i is a branch go to the next non-branch instruction at a position $i := i + k$ ($k > 0$).
      ii. Run Algorithm 3.1 until program position i.
      iii. Randomly select an effective destination register $r_{dest}(i) \in R_{eff}$.
4. If $l(gp) > l_{min}$ and (*deletion* or $l(gp) = l_{max}$) then
   (a) If *effective mutation* then select an effective instruction i if existent.
   (b) Delete instruction i.

5.10.3 Micro Mutations

Macro variations control program growth by operating on instruction level. While macro variation points only occur *between* instructions micro mutation points fall on a single instruction component, i.e., micro mutations operate *inside* instructions or on *sub*-instruction level. In all recombination-based and mutation-based LGP approaches of this chapter micro mutations are applied to replace single elements of instructions.

In Algorithm 5.5 three basic types of micro variations are distinguished – including operator mutations, register mutations or mutation of constants. Unless otherwise stated we mutate (exchange) each instruction component for about the same probability. In particular, this is true for destination registers and operand registers. The modification of either register position may affect the effective status of preceding instructions. As mentioned above register mutations correspond to redirections of edges in the functional representation of a linear program. That is, they manipulate the data flow in linear programs.

Constants may be replaced either by a register or by another constant depending on the proportion of instructions p_{const} that hold a constant value. Throughout this thesis we allow a constant to be set only if there is another register operand used by the same instruction (see also Section 6.3). This is an instruction may not hold more than one constant. Alternatively, separate constant mutations may be applied if $p_{constmut} > 0$ is true. Then a constant is selected explicitly from an instruction before it is modified through a standard deviation (step size) d_{const} from the current value.

Algorithm 5.5 (*(effective) micro mutation*)
Parameters: mutation rates for registers p_{regmut}, operators $p_{opermut}$, and constants $p_{constmut}$; rate of instructions with constant p_{const}; mutation step size for constants d_{const}.

1. Randomly select an [effective] instruction.
2. Randomly select mutation type *register* | *operator* | *constant* for probability p_{regmut} | $p_{opermut}$ | $p_{constmut}$ and $p_{regmut} + p_{opermut} + p_{constmut} = 1$.
3. If *register mutation* then
   (a) Randomly select a register position *destination* | *operand*.
   (b) If *destination* register then select a different [effective] destination register [using Algorithm 3.1].
   (c) If *operand* register then select a different *constant* | *register* for probability p_{const} | $1 - p_{const}$.
4. If *operator mutation* then select a different instruction operator randomly.
5. If *constant mutation* then
   (a) Randomly select an [effective] instruction with a constant c.
   (b) Change constant c through a standard deviation d_{const} from the current value: $c := c + \mathcal{N}(0, d_{const})$.

Since we guarantee for each genetic operator that there is a structural variation of the program code at all, identical replacements of code elements are avoided explicitly during micro mutations by Algorithm 5.5. As noted above, there is no *exchange* of instructions practiced with macro mutations.

5.10.4 Effective Instruction Mutations

When using macro mutations that change a single instruction only, more variations will become fitness-neutral, on average. Because mutation step sizes are small, many mutations stay noneffective, i.e., do not alter the structural effective code. To compensate this we introduce *effective instruction mutations* that avoid noneffective variations explicitly by concentrating mutations on the effective parts of a linear genetic program. This is motivated by the assumption that mutations of effective instructions may be less likely invariant (neutral) in term of a fitness change.

Effective mutations respect the functional structure of a linear genetic program (see Section 3.3) such that only the effective graph component is developed. In doing so, information about the functional program structure is introduced into the genetic operator.

The amount of noneffective code may be affected indirectly only through deactivations of depending instructions, i.e., disconnection of effective subgraphs.

We consider different approaches to *effective mutation* operators. The three approaches discussed mostly differ in terms of the way effective macro mutations are realized. Effective micro mutations simply select an effective instruction in Algorithm 5.5. If this does not exist, the destination register of a random instruction may be set effective.

One variant (effmut2) guarantees that both inserted and deleted instructions always alter the effective code. This includes that noneffective instructions are not selected explicitly for variation. The standard variant of effective mutations (effmut) allows (single) noneffective instructions to be deleted. In order to guarantee that the effective code is altered an effective (micro) mutation may directly follow such intron deletions. However, this may result in further deactivations of depending instructions and, thus, in more noneffective code. By allowing pure intron deletions, instead, the noneffective code is definitely reduced in the course of the evolutionary process compared to variant effmut2 (see below).

The explicit deletion of an effective or noneffective instruction is not complicated. Since the information about the effectiveness or non-effectiveness of an instruction is saved and updated in the linear program representation each time before the fitness calculation, no additional application of Algorithm 3.1 is necessary for effective micro mutations or effective deletions. After intron deletions the effective status does not have to be recalculated. Only after an effective variations the effectiveness of program instructions has to be verified.

If an instruction is inserted that is supposed to be effective afterwards, on the other hand, this has to be assured explicitly (see Algorithm 5.4). In particular, its destination register is chosen such that the instruction becomes effective at its position in the program. The choice of the operand registers is free. Recall from Definition 3.3 that a register is effective at a certain program position if its manipulation can effect the output of a program. Like the detection of effective code, effective registers can be identified efficiently in linear runtime $O(n)$. This is done by stopping Algorithm 3.3 at a certain program position i. Then set R_{eff} holds all registers that are effective at that position. An insertion of a branch instruction automatically becomes effective if the next non-branch instruction is effective. Otherwise, the destination register of this assignment is exchanged by an effective one.

If the program length is minimum only an insertion is possible. Accordingly, if the program length is maximum a deletion is applied next in Algorithm 5.4. Alternatively, an insertion might be allowed to replace another (effective) instruction in the latter case. But then the absolute step size is increased. It is not a feasible alternative to replace always a noneffective instruction by an effective one. This would definitely restrict the free choice of the mutation point because the rate of noneffective code may be quite small when using effective mutations. It must be noted, however, that this situation hardly occurs (see Section 5.11). Since programs grow relatively slowly by effective mutations the maximum program length may easily be chosen sufficiently large such that it is not reached within the observed number of generation.

There is only one situation where an effective deletion or insertion is not changing the effective code. This is the case if an instruction that is identical to the deleted/inserted one becomes effective/noneffective at the same position in the *effective* program. However, since this situations occurs only very rarely, it may be neglected.

In a third approach (effmut3) *all* emerging noneffective instructions are deleted directly after applying mutations of variant effmut2. If this would be done after free mutations, it is only guaranteed that deletions and micro mutations are effective. However, if instructions are not inserted effectively code growth might be too much restricted by a substantial loss of genetic material. By removing the structurally noneffective code completely linear GP becomes more similar to tree-based GP where such (disconnected) code does not exist because each node must be connected.

5.10.5 Minimum Effective Mutations

We implemented mutations on macro level and on micro level that induce a minimum change of the linear program structure. That is, the absolute mutation step size is minimum, i.e., comprises one instruction for macro mutations and one instruction component, i.e., one register, constant or operator, for micro mutations. Effective mutations assure that the (structurally) effective code is changed. However, it is not possible to predict for an (effective) mutation how many depending effective instructions will be deactivated or how many noneffective instructions will be reactivated afterwards. That is, these genetic operators do not explicitly guarantee that a certain effective variation step size (Definition 5.4) is met.

Minimum effective mutations reduce the effective variation distance between parent and offspring implicitly to the minimum. For micro mutations this means step size 0, i.e, no program instruction (above the mutated one[3]) is allowed to change its effectiveness degree. For macro mutations this is postulated for all instructions except for the inserted or deleted one, i.e. the minimum step size is 1. Variation operators have to select both the (effective) mutation point and the mutated code in such a way that no preceding program instruction is deactivated or reactivated. To achieve this information about the functional/data dependences within a linear genetic program may be used. We have demonstrated in Section 3.3 that linear genetic programs may be described by a directed acyclic graph (DAG). Minimum effective mutations only change one contiguous component of the graph, namely the effective one, while not allowing code to become non-contiguous.

Even if the choice of mutation point is free it would be unnecessarily complicated and computationally expensive to calculate a minimum effective mutation deterministically. Especially, full instruction mutations would require many register dependencies to be observed simultaneously and the effects of many potential mutations to be checked in advance. Instead, a minimization of effective step sizes may be better achieved by a probabilistic trial-and-error approach. This differs from an algorithmic calculation such that the effective mutation step size is meassured explicitly *after* a random mutation by means of a structural distance metric. A mutation is simply repeated then until a desired maximum distance is met.

It is important to note that the probabilistic induction does not increase the number of fitness evaluations. Only the structural step size has to be recalculated during each iteration which, however, does not require more than linear costs. The whole probabilistic induction of minimum step sizes will turn out to be runtime-efficient, because the probability decreases over a run that more than one trail is needed (see Chapter 8).

Actually, the effective code is able to protect itself by a increasing robustness against larger deactivations of effective code. We will see in Section 8.7.2 that this is mostly due to the number of usage connections between instruction registers that increase over a run. As a result, effective step sizes are already quite small, on average, when using normal effective mutations (implicit reduction of effective step sizes).

[3]Instructions below the mutation point cannot be affected.

5.10.6 Free Mutations

If we allow both noneffective and effective mutations to occur without imposing any kind of restriction this will be referred to as *free mutations* or *random mutations* (abbr. mut). In the last sections we have discussed operators that guarantee a modification of the effective code. Such code-effective mutations reduce the number of neutral variation effects significantly compared to random mutations. That is, more variations become constructive or destructive on fitness level accordingly. In general, the vast majority of non-neutral fitness changes is destructive (see Section 5.11.1).

When comparing effective and free mutations on the basis of generations, the effective variant is usually superior because evolution may progress faster within the same period of time. This is true because with the free variant the resulting number of effective operations is significantly lower and depends strongly on the ratio of effective and noneffective instructions in programs.

As mentioned in Section 5.2 the fitness does not have to be recalculated after noneffective variations since those are definitely neutral in terms of a fitness change. In other words, only effective variations cause (relevant) computational costs. Thus, if we compare both mutation variants after the same number of evaluations, but evaluate offsprings from effective variations only, the comparison becomes fair in terms of the computational overhead. Note that detecting the effectiveness or non-effectiveness of a variation *after* it has been executed requires the application of Algorithm 3.1 just as it is necessary for inducing effective mutations directly through the mutation operator. In this way, both mutation variants fall back on the information of where the effective code is located. It has to be considered, however, that the absolute (not the effective) variation step size is larger with the free variant, on average, because several (noneffective) mutations may happen between two fitness evaluations.

5.10.7 Explicit Induction of Neutral Mutations

The effective mutation approach has been introduced in Section 5.10.4 to increase the rate of *non-neutral* variations implicitly. Another interesting approach, that can give insight into the meaning of neutral variations in linear GP, may do exactly the opposite. The neutrmut operator transforms (most) destructive mutations into neutral or constructive ones. Therefore, it controls the direction explicitly in which the fitness of an individual is changing after a variation. The probabilistic control mechanism simply repeats an instruction mutation (mut) as long as it is destructive. Only after a maximum number of $n_{maxiter} > 1$ trails (iterations) a destructive variation is tolerated. Before each iteration the original state of the parent individual is restored. Offsprings from non-accepted variations are not exposed to evolution, of course. Each iteration produces extra computational costs in form of an additional fitness calculation. Only if the final variation is noneffective an evaluation is redundant and may be saved. The case $n_{maxiter} = 1$ corresponds to applying standard mutations.

Creating a high proportion of offsprings that result from a neutral variation may be expensive in terms of the number of evaluations. Usually more than one fitness calculation is necessary, on average, until an offspring is accepted to become part of the population. On the other hand, most neutral variations do not alter the effective code, i.e., are noneffective (see Section 5.11.1). This arises the question why not increasing the rate of *noneffective (neutral) variations* (abbr. noneffmut) directly. A probabilistic control may execute the mutation first and verify its effectiveness status afterwards. This is repeated until either a mutation is noneffective or a maximum number of iterations has been exceeded. Whether a variation is effective or noneffective can be verified efficiently since it requires a structural program analyses by Algorithm 3.1 only. This reduces the control of a semantic variation effect (neutrality) to the control of a structural variation effect.

A deterministic calculation of noneffective mutations, as applied for effective mutations in Section 5.10.4, is not practiced here. One reason is that, in general, non-effectiveness of variations is more complicated to guarantee than effectiveness, especially if the variation point is fixed beforehand. While in the latter case, only a single effective instruction has to be changed, in the former case, the effectiveness status of all instructions may *not* be changed.

Another reason originates from the fact that 100 percent noneffective variations do not make sense in any way, since the effective code as well as the fitness would never change. Instead, we are interested in an adjustable bias towards more noneffective variations. The maximum number of iterations $n_{maxiter}$ represents such a parameter.

Since the vast majority of neutral variations is noneffective, almost only the noneffective code is modified. To induce more neutral variations on the effective code (*effective neutral variations*) those must be controlled explicitly (abbr. neutreffmut). One way to achieve this is to apply the described probabilistic approach together with a certain percentage of effective mutations (effmut2 here). Recall that effective mutations are calculated deterministically (see Section 5.10.4). Increasing the probability of (effective) neutral mutations, instead, requires a trial-and-error method since it involves program semantics.

Not allowing mutations to become destructive may be regarded as an (1+1)EA selection [91] between parent and offspring. In an (1+1)EA the offspring only replaces the parent if its fitness is the same or better. This is different from a brood selection [94] where several offsprings of the same parent compete in a (tournament) selection process and only the winner gets into the population.

The reader may note that avoiding both destructive and neutral mutations in the same way, may not be a feasible alternative. Too many iterations and, thus, additional fitness evaluations would be necessary until an offspring is accepted and to increase the number of constructive variations significantly. Then a smaller maximum number of iterations would be almost always exceeded and a significant amount of variations would be still destructive. Moreover, the variation freedom may be too much restricted without neutral variations such that many intermediate variation steps are not possible only because they are not directly advantageous.

For a more general discussion about neutral variations we refer to Section 9.4 here.

5.11 Experiments II: Instruction Mutations

The different types of instruction mutations, which have been introduced in Section 5.10 above, are now compared with regards to the prediction performance and the (effective) size of solutions. Besides, the influence of certain control parameters on both criteria are in the center of interest. In particular, this includes the number and the distribution of mutation points as well as the use of an explicit grow bias.

5.11.1 Comparison of Instruction Mutations

The following eight Tables 5.19 to 5.26 compare the different approaches to mutation operators for all test problems from Section 5.8.1. The compared features comprise the *mean best* prediction error over 100 runs together with the statistical standard error (*std.*). Additionally, the number of hits is given, i.e., the number of times (from 100) how often

Operator	Config.	SSE		#Hits	Length			Variations (%)		
		mean	*std.*		*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
mut		6.5	0.3	0	78	32	41	0.5	63	62
noneffmut	maxiter 2	12.0	0.5	0	53	15	29	0.03	84	84
	maxiter 3	16.7	0.4	0	33	6	20	0.005	90	90
neutrmut	maxiter 2	5.4	0.3	1	84	38	45	0.3	81.5	80.9
	maxiter 3	6.0	0.3	0	87	42	48	0.2	89.4	88.6
neutreffmut	effmut 25%	3.7	0.2	0	98	52	53	0.8	70	68
	effmut 100%	**1.4**	0.2	14	60	37	62	13.1	15	0
effmut		2.2	0.2	16	29	24	80	8.2	9.4	4.9
effmut2		2.6	0.3	6	65	36	56	9.6	5.9	0
effmut3		1.9	0.2	15	23	23	100	9.3	6.4	0

Table 5.19: *distance*: Comparison of different (macro) mutation operators using bias configuration B1 for effective mutations and B0 otherwise. Average results over 100 runs after 1000 generations.

Operator	Config.	SSE		#Hits
		mean	*std.*	
mut		5.0	0.3	0
noneffmut	maxiter 2	6.3	0.3	0
	maxiter 3	6.2	0.3	1
neutrmut	maxiter 2	4.4	0.3	1
	maxiter 3	5.5	0.3	0
neutreffmut	effmut 25%	4.0	0.3	0
	effmut 100%	**2.7**	0.3	14

Table 5.20: *distance*: Comparison of different (macro) mutation operators using bias configuration B0. Average results over 100 runs after 1000000 (effective) evaluations.

the optimum has been found.

Absolute and effective program size are averaged over all individuals that occur during a run. Note that the size of best solutions remains almost identical to the average size of solutions. The reason is in the small standard deviation of (effective) lengths in the population that is smaller than 5 instruction. Both is a direct consequence from using minimum step sizes on the instruction level here.

Finally, we compare the distribution of variation effects, including constructive, neutral, and noneffective variations.

The results of the same runs are compared on the basis of two different measurements, generations and effective evaluations. In the first case, the number of new individuals in the population, i.e., all accepted variations, are regarded. In the second case, these are the *effective* variations only, but including the genetic operations that are *not accepted* during a neutrality control. The reader may recall from Section 5.2 that fitness is recalculated only if the effective code has been changed. Thus, a performance comparison on the level of effective evaluations better considers the computational costs. Nonetheless, comparisons on the level of generations are indispensable for experimental analyses concerning, e.g., program growth or variation effects. By abstracting from the computational costs of a

Operator	Config.	SSE		Length			Variations (%)		
		mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
mut		3.5	0.5	140	60	43	0.8	54	52
noneffmut	maxiter 2	8.6	1.0	146	59	40	0.2	80	79
	maxiter 3	17.6	1.4	131	39	30	0.02	86	86
neutrmut	maxiter 2	1.4	0.2	154	76	49	0.6	72	70
	maxiter 3	1.5	0.2	158	83	53	0.6	82	80
neutreffmut	effmut 25%	0.9	0.11	154	82	53	1.0	66	63
	effmut 100%	**0.3**	0.03	82	58	71	9.8	22	0
effmut		0.9	0.06	39	33	85	6.9	14	3.6
effmut2		1.0	0.06	57	39	69	7.6	12	0
effmut3		1.1	0.07	27	27	100	7.8	11	0.1

Table 5.21: *mexican hat*: Comparison of different (macro) mutation operators using bias configuration B1. Average results over 100 runs after 1000 generations.

Operator	Config.	SSE	
		mean	*std.*
mut		2.3	0.4
noneffmut	maxiter 2	3.9	0.5
	maxiter 3	4.5	0.5
neutrmut	maxiter 2	1.2	0.4
	maxiter 3	1.4	0.2
neutreffmut	effmut 25%	1.1	0.13
	effmut 100%	**0.6**	0.06

Table 5.22: *mexican hat*: Comparison of different (macro) mutation operators using bias configuration B1. Average results over 100 runs after 1000000 (effective) evaluations.

variation and by comparing evolutionary progress after the same number of newly created solutions in the population, we may obtain fundamental knowledge that not only gives us a better understanding of GP but can be valuable for designing more efficient genetic operators.

The results obtained with effective mutations (effmutX) are given only for one unit of time measurement. Depending on the implementation the rate of noneffective variations is very small or zero with this operator. Thus, results after 1000 generations or 1000000 effective evaluations (with population size 1000) differ only very slightly or not at all. In general, the performance of a genetic operator is the more similar with both measurements the less noneffective variations it produces and the less variations are rejected during a neutrality control (if used). The effective mutation operator implicitly increases the rate of non-neutral variations including a higher rate of both destructions and constructions while destructions are by far the most dominating variation effect. About 85 percent of all variations are destructive with the tested approximation problems and about 65 percent with the classification problems.

All three different variants of effective mutation operators (see Section 5.10.4) work almost equally well here. Little differences may result either from a slower growth of (effective) code due to a radical removal of all noneffective instructions (effmut3) or from a faster

Operator	Config.	CE		#Hits	Length			Variations (%)		
		mean	*std.*		*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
mut		15.5	0.6	1	132	57	43	0.2	62	49
noneffmut	maxiter 2	37.6	2.3	0	134	39	29	0.03	87	83
	maxiter 3	68.4	3.1	1	124	24	19	0.007	96	95
neutrmut	maxiter 2	13.4	0.7	2	142	65	46	0.1	82	64
	maxiter 3	10.5	0.6	2	143	70	49	0.1	90	68
neutreffmut	effmut 25%	8.4	0.5	3	143	92	64	0.1	84	41
	effmut 100%	**5.9**	0.4	10	126	110	87	0.4	72	0
effmut		13.9	0.7	2	77	71	92	1.1	38	1.9
effmut2		12.1	0.7	5	96	84	87	1.0	39	0
effmut3		14.0	0.7	1	63	63	100	1.4	34	0

Table 5.23: *three chains*: Comparison of different (macro) mutation operators using bias configuration B1. Average results over 100 runs after 1000 generations.

Operator	Config.	CE		#Hits
		mean	*std.*	
mut		11.8	0.6	1
noneffmut	maxiter 2	13.3	0.6	1
	maxiter 3	12.3	0.7	4
neutrmut	maxiter 2	11.8	0.6	2
	maxiter 3	9.9	0.6	3
neutreffmut	effmut 25%	**9.3**	0.6	1
	effmut 100%	10.5	0.6	2

Table 5.24: *three chains*: Comparison of different (macro) mutation operators using bias configuration B1. Average results over 100 runs after 1000000 (effective) evaluations.

growth due to a higher proportion of such instructions (effmut2). The effmut2 variant demonstrates that the noneffective code remains small even if deletions of noneffective instructions are not allowed explicitly (as with effmut). Note that the rate of noneffective variations equals the rate of such intron deletions since all other variations are effective. Depending on the correlation between problem fitness and program length different variants may be superior. For instance, variant effmut2 works better here with the classification problems, *three chains* and *spiral*.

The effmut3 results show that the existence of structural introns in linear genetic programs is less important, at least for the performance of effective mutations. The multiple register usage, i.e., the graph-based data flow, in linear programs allows the effective code to protect itself sufficiently against larger deactivations and, thus, against the loss of code (see also Chapter 8). However, the use of an explicit grow bias (B1 here) becomes more important with this variant of effective mutations (see also Sections 5.11.3). This compensates partly the loss of genetic material by a faster code growth.

Effective mutations perform better than free mutations (mut) if the same number of variations (generations) is regarded. On the level of effective variations (evaluations), however, random mutations may perform equally well or even better than mutations that vary the effective code exclusively. This situation occurs here with the two classification problems

Operator	Config.	CE		#Hits	Length			Variations (%)		
		mean	*std.*		*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
mut		13.6	0.6	0	128	64	50	0.3	50	42
noneffmut	maxiter 2	18.0	0.6	0	139	60	43	0.03	75	72
	maxiter 3	25.5	0.8	0	135	50	37	0.005	89	87
neutrmut	maxiter 2	8.7	0.4	0	143	79	56	0.1	70	57
	maxiter 3	6.0	0.3	1	148	83	56	0.1	83	67
neutreffmut	effmut 25%	2.9	0.2	13	148	101	68	0.2	70	41
	effmut 100%	**2.3**	0.2	20	120	109	91	0.8	55	0
effmut		8.8	0.4	2	74	69	93	1.7	24	2
effmut2		7.2	0.5	1	86	77	90	1.4	25	0
effmut3		9.0	0.4	0	56	56	100	1.9	22	0

Table 5.25: *spiral*: Comparison of different (macro) mutation operators using bias configuration B1. Average results over 100 runs after 1000 generations.

Operator	Config.	CE		#Hits
		mean	*std.*	
mut		9.0	0.4	0
noneffmut	maxiter 2	9.0	0.4	0
	maxiter 3	10.5	0.4	0
neutrmut	maxiter 2	8.4	0.4	0
	maxiter 3	6.7	0.3	1
neutreffmut	effmut 25%	**5.7**	0.3	2
	effmut 100%	7.1	0.4	5

Table 5.26: *spiral*: Comparison of different (macro) mutation operators using bias configuration B1. Average results over 100 runs after 1000000 (effective) evaluations.

and may result directly from a higher rate of noneffective neutral variations or indirectly from a larger size of solutions. A faster code growth has turned out to be more important for the discrete test problems than this has been found with the continuous ones, *distance* and *mexican hat*.

The neutrmut approach applies an explicit control of neutral variations as introduced in Section 5.10.7. After a variation is accepted or a maximum number of iterations (2 or 3 here) has been exceeded the offspring is copied into the population. Otherwise, the operation is repeated. Thus, one variation step may require more than one fitness evaluation, on average. This makes a comparison on the basis of evaluations necessary. The neutrality control increases the rate of neutral variations up to about 90 percent here. If we compare the rate of noneffective variations we can see that almost all neutral variations are noneffective, too, as far as the approximation problems are concerned. By comparison, for the classification problems the proportion of noneffective variations is definitely smaller. On the one hand, neutral variations that alter the (structurally) effective code (see Section 5.10.7) are induced more easily here because discrete fitness functions facilitate the propagation of semantic introns. On the other hand, effective programs grow by branches because these allow larger semantic introns and a higher specialization to the training data.

Since about half of the variations turns out to be noneffective (and thus neutral) already with the standard approach (mut), the neutrality control may affect at most 50 percent of variations only that would be destructive, otherwise. Recall that noneffective variations do not produce computational costs in terms of fitness evaluations. Besides, we found that already about 2 trials are sufficient, on average, to achieve that almost all mutations become neutral. Hence, the number of necessary fitness evaluations is only doubled compared to the standard approach. In other words, only about the same total number of fitness evaluations is required for promoting neutrality of variations as this is necessary for avoiding neutrality (effmut).

Concerning the prediction quality Tables 5.19 to 5.26 document that most test problems profit from an explicit induction of more neutral mutations (neutrmut). One important argument for this is the higher survival probability of individuals resulting from a neutral variation (see Chapter 11). In general, improvements in prediction error (compared to standard mutations) are more significant on a generation basis than on an evaluation basis here.

One question was whether similar improvements may already be obtained by simply increasing the rate of noneffective variations. Recall that a verification of non-effectiveness does not require additional fitness calculations. Unfortunately, the noneffmut series demonstrates that, by only increasing the rate of noneffective neutral variations, the prediction error is decreased drastically on a generational basis. A too low rate of effective variations leads to a too low rate of constructive operations and, in some cases, to a smaller effective size of programs, too. When comparing results after the same number of effective evaluations, this disadvantage is partly compensated. But the performance is still worse than it is achieved with standard mutations. Note that the total variation step size increases significantly here because of the high number of noneffective mutations (only one effective) that may happen between two fitness evaluations. Consequently, if a higher rate of noneffective variations does not improve solution finding, the (slightly) larger difference between the proportions of neutral and noneffective variations that occurs with the neutrmut operator seems to be essential.

In order to increase the rate of such effective neutral variations more explicitly the neutreffmut approach applies a neutrality control together with (a certain percentage of) effective mutations. Interestingly, this combination improves performance compared to applying both approaches separately, especially on the basis of generations. On the basis of effective evaluations, however, results may be similar to the results obtained already when using the effective mutation operator alone (compare effmut2 here). With the continuous test problems the rate of noneffective variations and the rate of neutral variations decrease at almost the same amount. The rate of constructive variations is similar as (or even higher than) with normal effective mutations.

With the two discrete problems, instead, less neutral variations are noneffective, too. Interestingly, even if effective mutations are applied for 100 percent (neutreffmut) the resulting rate of neutral variations decreases only slightly. But already 25% explicitly induced effective mutations let effective neutral variations occur significantly more frequently here Obviously, the maximum number of 3 iterations is not exceeded very often. We will argue in Chapter 9 that the induction of effective neutral variations is strongly correlated with the ability of a problem configurations to create semantic introns (see Definition 3.2). Moreover, these variation effects seem to be highly advantageous during evolution.

On a generational basis the neutreffmut operator achieves a much higher gain in performance than the neutrmut operator. An explicit control of effective neutral variations is, however, more expensive in terms of the number of necessary fitness evaluations. Hence, on the basis of evaluations the difference in average prediction error shrinks between neutreffmut and neutrmut. Nevertheless, except for the *distance* problem the performance is still better than this has been found by using effective mutations (effmutX) only.

In general, we may conclude that increasing the proportion of both neutral *and* effective mutations actively results in the highest gain in performance for all test problems. Smaller absolute and effective solutions, however, are achieved by using standard effective mutations only which are mostly destructive. Chapter 9 will demonstrate that a small noneffective code is a direct result from the low rate of noneffective (neutral) variations. Correspondingly, the effective code grows larger with effective neutral variations than with destructive variations.

5.11.2 Comparison with Segment Variations

A comparison between free instruction mutations here and segment mutations (onesegmut in Section 5.9.2 reveals a significantly better performance in favor of the first approach (for *mexican hat* and *spiral*). This results mostly from the minimum step size of instruction mutations rather than from a smaller size of (effective) solutions which differs only slightly here. Recall that segment mutations have been configured with a maximum (unlimited) step size.

Figure 5.8 shows the fitness progress of the currently best individual over the generations for different macro operators. A lot of information is gained at the beginning of a GP run. During this period (best) fitness improves most significantly. Towards the end of a run the (absolute) fitness improvements become smaller. In other words, the convergence speed of the fitness decreases over a run.

First, one can see that (effective) instruction mutations perform better than crossover already from the beginning of a run. The larger absolute step sizes of crossover do not seem to be more successful in early generations. Second, the differences in fitness values do not change much here between the operators in the last 500 generations.

In particular, the difference between effective and free instruction mutations does not necessarily decrease towards the end of a run. The effectiveness of random mutations – including insertions and deletions – depends on the ratio of effective code and noneffective code in programs. This ratio stays more-or-less constant during a run as long as the size of programs has not reached the maximum limit (not shown). In this case, the effective code may still grow even if this happens more slowly (as shown in Figure 5.9).

5.11.3 Explicit Grow Bias

By using macro mutations with a minimum step sizes of one instruction the (maximum) speed of code growth is restricted most. Therefore, we will test the influence of different grow biases (see Section 5.8) on the performance of (effective) instruction mutations. Depending on the proportion of insertions and deletions of instructions, the speed of code growth may either be affected positively (grow bias) or negatively (shrink bias). If insertions and deletions are applied for the same probability there is no such bias of the mutation operator defined explicitly. Basically, the speed with which programs may grow during a certain number of generations depends on both the problem and the macro variation operator. While the problem definition determines the correlation between the (effective) solution size and the fitness, an *explicit bias* of the variation operator is semantically independent. In contrast to an *implicit bias* it will influence code growth even without fitness information (see also Chapter 9).

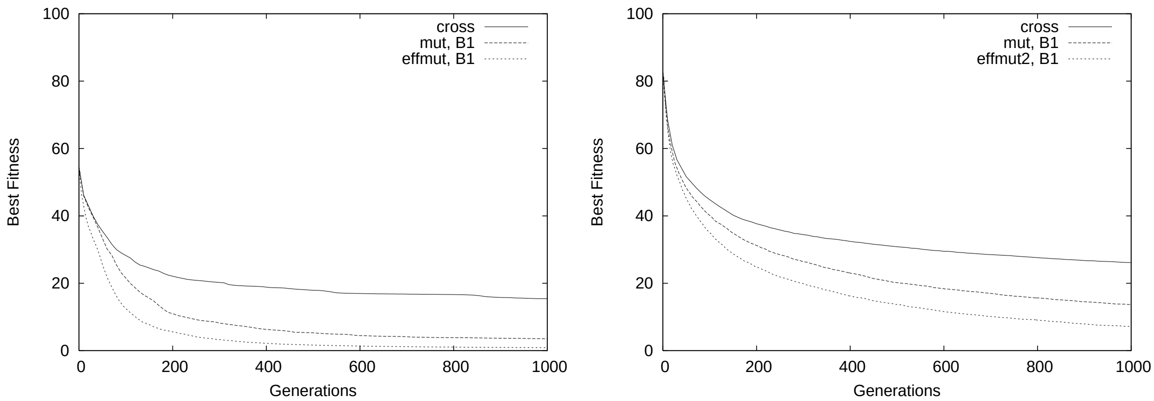

Figure 5.8: Development of best fitness for different (macro) variation operators with *mexican hat* (left) and *spiral* (right). Average figures over 100 runs.

In Tables 5.27 and 5.28 the influence of different bias configurations on the best prediction performance and the average program length is compared. First of all, for the same bias configuration the absolute program lengths are usually smaller with effective mutations (effmutX) than with standard mutations (mut). The main reason is a much lower rate of noneffective code that emerges if (almost) only effective code is changed or is created newly during variation. The relative difference in program length becomes smaller with stronger grow biases due to the maximum length bound (200 instructions here).

Operator	Config.	SSE		Length			Variations (%)		
		mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
mut	B–1	1.7	0.2	37	25	68	1.9	37	35
	B0	2.4	0.3	72	41	58	1.3	45	43
	B1	3.5	0.5	140	60	43	0.8	54	52
	Bmax	6.9	0.9	179	75	42	0.8	55	53
effmut	B0	1.3	0.09	26	23	88	7.0	13	4.2
	B1	0.9	0.06	39	33	85	6.9	14	3.6
	Bmax	0.9	0.06	101	72	71	7.3	14	0.6
effmut3	B1	1.1	0.07	27	27	100	7.8	11	0
	Bmax	0.6	0.05	54	54	100	7.3	12	0

Table 5.27: *mexican hat*: Comparison of free mutations and effective mutations with different bias configurations. Average results over 100 runs after 1000 generations.

For the same bias configuration the average program size remains similar for the different test problems when we apply standard instruction mutations. Interestingly, this is true for the effective size as well. Effective mutations, instead, allow solution sizes to differ more strongly between problems since less noneffective code occurs with these variations. Then the program length is more subject to the fitness selection.

The proportion of noneffective code may increase slightly together with the insertion rate (bias). This may be interpreted as a protection reaction of the system to the higher rate of instruction insertions and the resulting higher growth of (effective) code. For larger biases this effect is weakened by the influence of the maximum program length (see below). Another reason is the reduced ability of some problems and function sets to develop semantic introns (see Table 5.27).

The average prediction error in Table 5.27 documents a clear negative influence of a positively biased program growth when using free mutations. Instead, the *mexican hat* problem is solved best with a shrink bias. The tested shrink bias B–1 reduces absolute and effective code growth almost by half compared to the bias-free configuration B0.

Operator	Config.	CE		#Hits	Length			Variations (%)		
		mean	*std.*		*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
mut	B0	15.0	0.5	0	75	44	60	0.5	42	36
	B1	13.6	0.6	0	128	64	50	0.3	50	42
	Bmax	13.4	0.6	0	176	88	50	0.2	52	42
effmut2	B0	11.6	0.4	1	55	50	91	2.1	21	0
	B1	7.2	0.4	1	86	77	90	1.4	25	0
	Bmax	6.4	0.3	3	155	136	88	1.1	30	0
effmut3	B1	9.0	0.4	0	56	56	100	1.9	22	0
	Bmax	5.3	0.3	1	122	122	100	1.7	23	0

Table 5.28: *spiral*: Comparison of free mutations and effective mutations with different bias configurations. Average results over 100 runs after 1000 generations.

In contrast to free mutations, grow bias B1 has always been found to improve the performance of the effective mutation operator. The maximum grow bias Bmax, however, has not turned out to be much more successful than bias level B1, but produces significantly larger solutions only. Only the effmut3 variant is still improved clearly if insertions of instructions are applied exclusively. Actually, the effmut3 operator performs best then. Recall that programs grow more slowly here due to a radical deletion of introns (see Section 5.10.4).

Figures 5.9 and 5.10 illustrate exemplarily the development of absolute and effective length over the generations for free and effective mutations. Note that the influence of an explicit bias on code growth is relaxed as soon as a genetic program has reached its maximum size. In this case, only instruction deletions are possible (see Algorithm 5.4). Thus, if only insertions are applied otherwise (Bmax) the rate of insertions and deletions is almost balanced for such programs. This corresponds to applying no bias at all and affects both the absolute program length and the effective length. Also note that the growth of effective code is decelerated in Figures 5.9 as soon as the average absolute size approaches the maximum.

We conclude with some more general considerations about applying an explicit grow bias in genetic programming. To keep structural mutation steps permanently small between the fitness evaluations it is required that these are possible at almost all positions of the representation. In other words, the variability of the representation must be sufficiently high. This is mostly true for the linear representations and its graph-structured data flow (see Section 5.10.1). Otherwise, a grow bias may be implemented only such that smaller subprograms are replaced by larger ones for a higher probability. This, however, implies larger structural changes, too. In the following section we will demonstrate that a grow bias – in combination with a minimum mutation step size – may not be outperformed by using larger step sizes in form of multiple mutations.

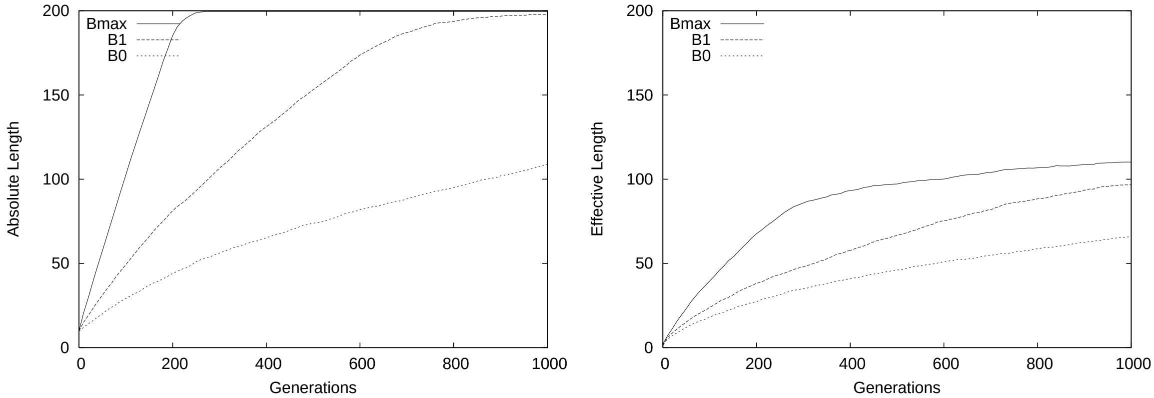

Figure 5.9: *mexican hat*: Development of absolute program length (left) and effective program length (right). Influence of different grow biases on free mutations (mut). Average figures over 100 runs.

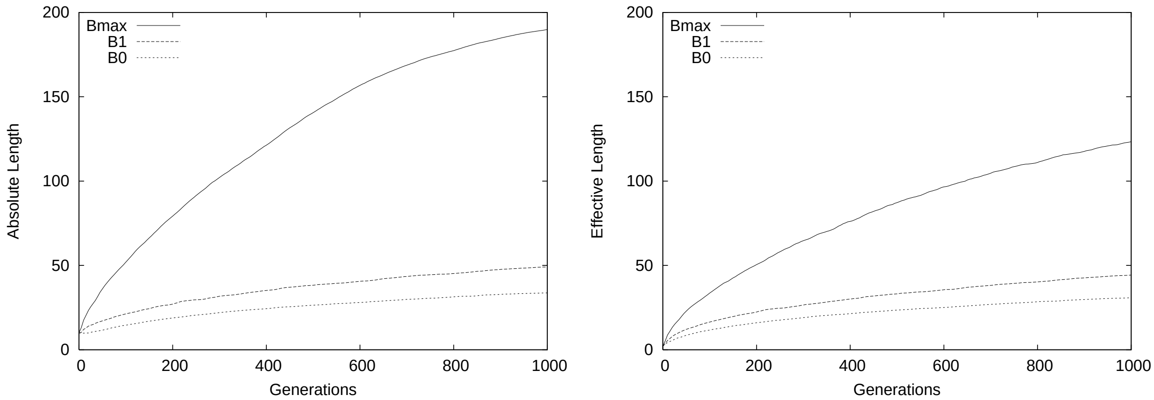

Figure 5.10: *mexican hat*: Development of absolute program length (left) and effective program length (right). Influence of different grow biases on effective mutations (effmut). Average figures over 100 runs.

5.11.4 Number of Mutation Points

In the experiments documented above we have introduced a bias into the mutation operator to control the growth of programs more explicitly. In doing so, the evolutionary process may be guided faster to regions of the search space where the complexity of solutions is suitable for finding the optimum or a good suboptimum solution. We have seen in the previous section that, depending on the problem as well as on the considered number of generations, this may require code growth to be accelerated or decelerated.

Provided that a problem fitness profits from a faster growth of programs, it might be argued that a biased operator is not really necessary. Instead, program growth might be accelerated by allowing larger absolute step sizes. We will demonstrate in the following that this is not absolutely true and that a minimum mutation step size yields the best performance.

The absolute mutation step size is controlled by the maximum number of mutations that may be applied to an individual simultaneously, i.e., without exposing the intermediate results to fitness selection. This number is selected uniformly distributed from a certain

Maximum	SSE		Length			Variations (%)		
#Mutations	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
1	**1.3**	0.1	39	27	70	8.1	10	0
2	1.7	0.1	38	24	63	8.8	11	0
5	2.6	0.2	53	28	53	9.2	14	0
10	3.5	0.2	76	35	46	9.2	15	0
20	7.8	0.4	102	44	43	8.6	16	0

Table 5.29: *mexican hat*: Multiple effective mutations (effmut2, B0). Average results over 100 runs after 1000 generations.

Maximum	SSE		Length			Variations (%)		
#Mutations	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
1	1.6	0.2	72	41	58	1.3	45	43
2	**1.2**	0.1	69	37	53	2.1	37	34
5	1.7	0.2	68	31	46	3.9	26	23
10	2.1	0.2	64	24	37	5.3	23	17
20	4.0	0.4	73	23	32	6.2	22	12

Table 5.30: *mexican hat*: Multiple mutations (mut, B0). Average results over 100 runs after 1000 generations.

maximum range here and is valid for both micro mutations and macro mutations. Note that the mutation type is selected only once before mutations of that type are applied as often as specified. Both deletions and insertions of instructions may happen in one variation step, instead.

Alternatively, structural mutation steps might be controlled over a maximum segment length. One basic difference to the approach applied here is that mutation points may be chosen less freely on the imperative level since all inserted or deleted instructions are necessarily arranged in a sequence. Then a segment of effective instructions more likely represents a single contiguous component in the functional representation. Another difference is that the insertion of a segment may affect less variation points of the program graph than the insertion of multiple effective instructions (at multiple positions).

Multiple effective mutations would actually require that the effective code (and the noneffective code) is redetermined after each partial mutation. However, it has not been found to make any difference in terms of prediction quality and code growth whether this is practiced or not. Nevertheless, it is applied here since the detection of effective instruction (see Section 3.2.1) is not computationally expensive for a moderate number of mutation points.

The optimum configuration comprises a single effective instruction only that is mutated, deleted, or inserted (effmut). The experiments documented in Tables 5.29 and 5.31 demonstrate this for both test problems, *mexican hat* and *spiral*. By using free mutations (mut), instead, the optimum number of mutation points may be larger, mostly because the whole variation becomes effective for a higher probability in this way. Nevertheless, only two instructions turned out to be optimum in Tables 5.30 and Table 5.32.

In general, effective mutations perform better than free mutations if the mutation rate (on the program representation) is smallest, because too many free mutations stay noneffective then. If many free mutations happen simultaneously, instead, the probability for a

Maximum #Mutations	CE mean	CE std.	#Hits	Length abs.	Length eff.	Length %	Variations (%) constr.	Variations (%) neutral	Variations (%) noneff.
1	**7.6**	0.4	2	86	78	91	1.7	25	0
2	10.4	0.5	0	81	71	87	2.6	19	0
5	16.4	0.5	0	79	63	80	4.8	14	0
10	21.8	0.6	0	80	59	73	6.0	15	0
20	28.5	0.6	0	88	58	66	6.3	20	0

Table 5.31: *spiral*: Multiple effective mutations (effmut2, B1). Average results over 100 runs after 1000 generations.

Maximum #Mutations	CE mean	CE std.	#Hits	Length abs.	Length eff.	Length %	Variations (%) constr.	Variations (%) neutral	Variations (%) noneff.
1	15.0	0.4	0	75	44	60	0.5	42	36
2	**13.9**	0.5	0	76	44	58	1.0	34	27
5	16.7	0.6	0	76	39	51	2.3	24	13
10	22.0	0.6	1	66	31	46	4.0	18	12
20	25.6	0.7	0	58	23	40	5.3	18	8

Table 5.32: *spiral*: Multiple mutations (mut, B0). Average results over 100 runs after 1000 generations.

noneffective variation is lower as well as the difference in error.

Interestingly, the average effective length decreases if more free mutations are applied simultaneously while the absolute length stays constant or decreases less (see Tables 5.30 and 5.32). The shrinking proportion of effective code may be interpreted as a protection reaction of the system to reduce the average *effective* step size (see Section 5.3). By means of a higher proportion of noneffective instructions single mutations are noneffective for a higher probability. Apparently, this is true for deletions of instructions. But also the effectiveness of random insertions depends at least in part on this proportion. A similar protection mechanism has been observed with crossover in Section 5.9.4. If numerous mutations happen simultaneously, non-effectiveness becomes more unlikely for the whole variation step. Hence, the proportion of noneffective variations decreases. Additionally, the effective code may be larger for smaller mutation steps because those allow a more precise approximation to better solutions.

If we induce effective mutations only (effmut2) the proportion of noneffective variations is zero. In Table 5.29 and Table 5.31 we can observe a reduction of the effective code rate, too, even if the amount of effective code grows with higher mutation rates in case of the *mexican hat* problem.

It is important to note in this context that an explicit grow bias (which has been used only for the experiment documented in Table 5.31 here) is not reinforced by using multiple mutations. These may not affect the ratio of inserted and deleted instructions. The reader may recall that the absolute variation step size does not influence code growth directly. By definition, it just determines the possible distance in length between parent and offspring during one variation step.

First, we may conclude that a larger than minimum mutation rate on the program representation works worse, at least, when using effective mutations. In other words, a fitness evaluation after each instruction mutation is essential and may not be saved. This shows that a minimum structural step size still induces semantic step sizes that are large enough, on average, to escape from local minima of the fitness landscape. Second, a higher mutation step size may not be regarded as an alternative to an explicit grow bias. Neither the prediction error improves by using several effective mutation points nor does the length of programs grow necessarily.

5.11.5 Self-Adaptation

Self-adaptation of variation parameters has been applied successfully in different disciplines of evolutionary algorithms [8]. In evolution strategies (ES) [91, 75] standard deviations of mutation step sizes are treated as part of the individual representation, i.e., a real-valued vector of objective values. In the most simple case there is only one mutation parameter (standard deviation) used for all objective variables. Rather than using a deterministic control rule for the adaptation of such parameters, the parameters themselves are subject to evolution. Self-adaptation differs from a global adaptive control of parameters in such a way that the parameters are adapted locally. The modification of the parameters is under the control of the user only by means of a fixed mutation step size (*learning rate*).

Selection is performed on the basis of the individual fitness only. The propagation or extinction of variation parameters in the population is coupled with the fitness of their carrier individuals. Consequently, the success of a certain parameter configuration is directly depending on how the variation operator performs on an individual when using these settings. It is generally recommended to mutate the variation parameters of an individual first before the new settings are applied for the variation of the individual. The reversed mechanism might suffer from a propagation of (good) individuals with rather bad parameter settings because those have not been used for finding the current position of the individual on the fitness landscape.

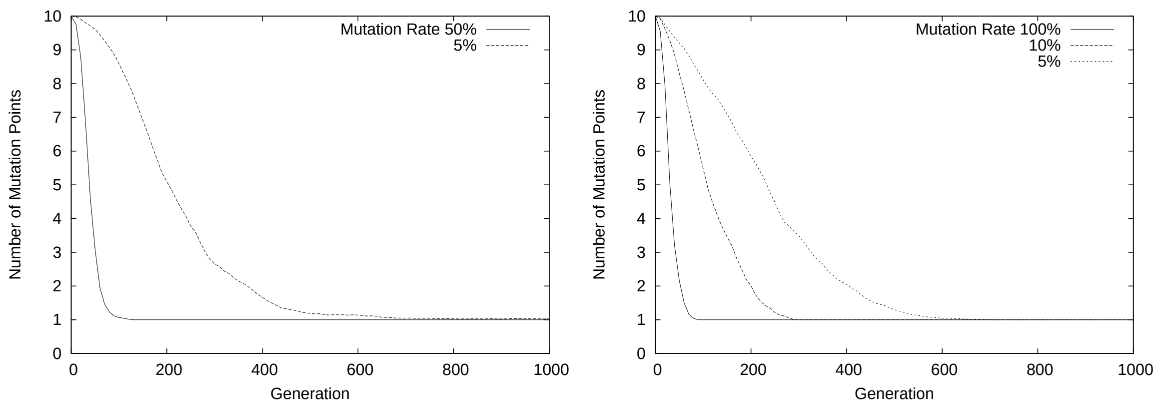

Figure 5.11: Development of the maximum number of mutation points with self-adaptation for different parameter mutation rates using *mexican hat* (left) and *spiral* (right). Numbers averaged over all parameter values in the population. Average figures over 100 runs.

Moreover, better results may be obtained when using a lower mutation rate (and a lower reproduction rate) for parameters than for individuals. Otherwise, good individuals with bad parameter settings might spread too quickly in the population at the beginning of a run. This again may lead to an earlier loss of diversity while the search process gets caught more easily in a local minima, at least in terms of the parameter space. Note that the fitness of an individual does not depend directly on the quality of its variation parameters. But the parameters influence the expected average fitness of its offsprings.

The motivation for a self-adaptation is twofold. On the one hand, it may outperform an optimum global setting that stays constant during a run because a variable (dynamic) setting of a mutation step size turns out to be more advantageous. On the other hand, even if this is not true self-adaptation may be applied for finding an optimum (or a nearly optimum) configuration. Especially if the dimension of the parameter vector is high, an optimum configuration may not be detected efficiently in general by trying constant settings.

The principle of self-adaptation is applied here for the coevolution of structural mutation steps in linear GP. This may either be the number of mutation points (instruction mutations) or the segment length (segment mutations). Here the number of effective instruction mutations is self-adapted. To achieve this, only one parameter has to be encoded into each program individual which is the *maximum* mutation step size n. The actual step sizes may be selected then either uniformly distributed or normally distributed over the maximum range by using an expectation value of 0 and a positive standard deviation of $p \times n$ $(0 < p \leq 1)$. We choose a uniform distribution here. Note that only non-zero positive integer values are defined as step sizes on the symbolic representation. Individual parameter values may become 0 but are mapped to 1 if applied as a step size. The variation of the individual parameter values is controlled by a mutation probability and a constant mutation step size of ± 1.

The results of the last section have shown clearly that the optimum performance is obtained with the minimum number of effective mutation points (one). Even if a varying number of effective mutation points during runtime may still turn out to be more successful, such a result allows only a small range of improvements. Nevertheless, self-adaptation may provide information about how precisely and how fast a (constant) optimum is approximated.

Besides, such experiments may provide information about whether a higher mutation step size may have a positive influence at the beginning of a run. In general, this is motivated by a higher diversity in the initial generations which makes the evolutionary algorithm less depending on the composition of the genetic material in the initial population.

Figure 5.11 shows how the mutation step size develops over the generations when using self-adaptation. As one can see the average individual step size in the population converges to the minimum step size 1 for both problems, *mexican hat* and *spiral*. The higher the mutation probability is set for the step size parameter the more quickly the minimum is reached. We have checked that no convergence occurs without fitness. In this case, the average mutation step size during a run oscillates around the value that has been provided initially.

The prediction performance (not shown here) comes very close to the performance that is obtained with constant step size 1 in Tables 5.29 and 5.31. That is, a varying (maximum) step size on the symbolic structure of individuals during runtime has not been found significantly better than using minimum step size 1 continuously. It appears that larger structural steps may not be more successful locally since they reduce the survival probability and the potential fitness of offsprings.

At least for the applied mutation-based configuration a higher mutation rate at the beginning of a run does not seem to be beneficial. It remains an open question, however, whether this is different in significantly smaller populations. Nevertheless, it is interesting to note that using a larger mutation step size at the beginning of a run has a less negative influence on the prediction performance than using a constant setting of 2 mutation points (maximum) over the whole run.

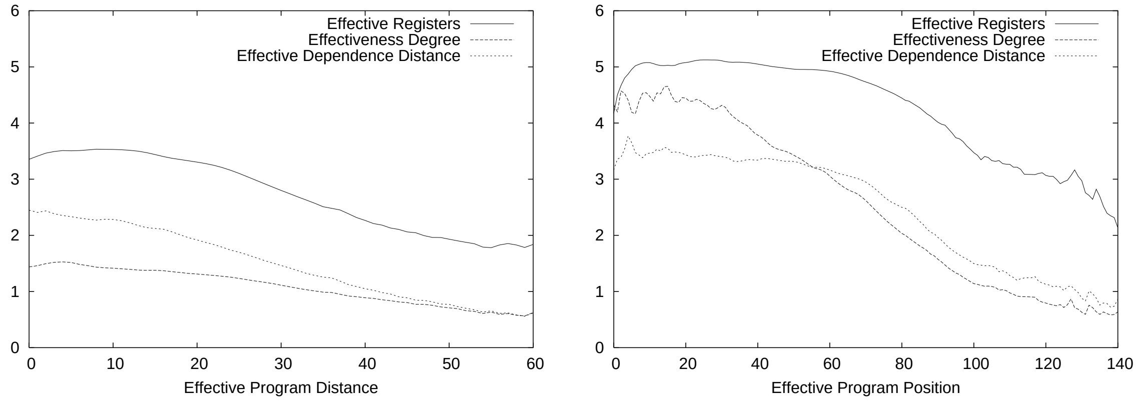

Figure 5.12: Development of the number of effective registers, the degree of effectiveness, and the effective dependence distance over effective program positions using effective mutations (effmut). Position 0 holds the first instruction of a program. Average figures over all programs of the final (1000th) generation and over 100 runs. Results for *mexican hat* (left) and *spiral* (right).

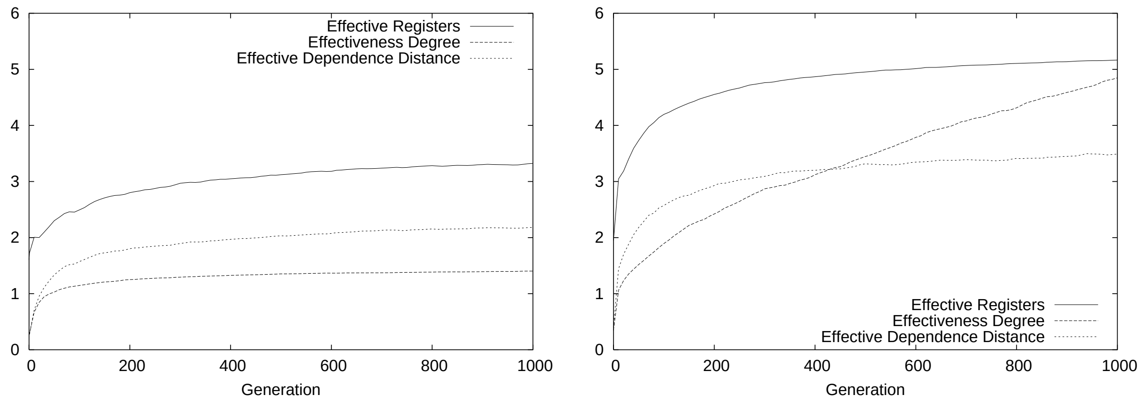

Figure 5.13: Development of the average number of effective registers, the average degree of effectiveness, and the average effective dependence distance over the generations using effective mutations (effmut). Average figures over 100 runs. Results for *mexican hat* (left) and *spiral* (right).

5.11.6 Distribution of Mutation Points

In the next series of experiments we investigate the choice of the mutation point. In the standard case each instruction is chosen for the same probability. But is such a uniform distribution of mutation points really close to the optimum ? At first sight, this might be true for an imperative representation that is composed of a linear sequence of instructions.

We learn from Figure 5.12 how the functional structure of a linear program is built by applying the three algorithms from Section 3.4. For each program position the structural information are averaged over all effective programs of the final generation that hold an instruction at that position. The average effective length is about 55 instructions for *mexican hat* and 110 instructions for *spiral*. The standard deviation of effective lengths in the final population is below 5 instructions.

Most important here is the information how the average number of effective registers and the average effectiveness degree develop over the effective instruction positions in Figure 5.12. Over the first half of the program length (from the beginning) the number of effective registers is more-or-less constant here. Only over the last half, it decreases until it becomes 1 at the last effective instruction in a program. (The average value is larger here due to variable program lengths.) Since both the effectiveness and the effective dependence distance are 0 at the last effective instruction, the average may become smaller than 1. The effectiveness degree of instructions as defined in Section 5.5 corresponds to the connection degree of nodes in the (effective) functional representation. We can see in Figure 5.12 that the effectiveness of instructions decreases more regularly towards the end of a linear program.

The reason for these observations becomes clear if we recall from Chapter 3.3 that the last effective instruction of a linear program corresponds to the root of the underlying (effective) graph component. The number of effective registers at a certain instruction position denotes an approximation of the graph width at the corresponding instruction node. It appears that this width grows quickly until a certain maximum (starting form the graph root) and stays rather constant then because it is restricted by the total number of available registers. We will demonstrate in Section 6.1 that the use of too many registers in linear genetic programs is not recommended in general. Among other things, a restriction is necessary in order not to increase the search space of programs unnecessarily. A wider graph requires a longer imperative representation. Correspondingly, the distance of depending (effective) instructions increases in Figure 5.12 together with the number of effective registers.

Mutation	SSE		#Hits	Length			Variations (%)		
Distribution	*mean*	*std.*		*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
$\mathcal{U}(n)$	2.3	0.2	16	29	24	80	8.2	9	4.9
$\lvert\mathcal{N}(0, 0.33n)\rvert$	2.3	0.2	3	39	30	78	7.6	11	5.1
$n - 1 - \lvert\mathcal{N}(0, 0.33n)\rvert$	7.0	0.2	0	39	26	67	8.3	12	6.3

Table 5.33: *distance*: Comparison of different frequency distributions of effective mutation points.

Mutation	SSE		Length			Variations (%)		
Distribution	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
$\mathcal{U}(n)$	0.9	0.06	39	33	85	6.9	14	3.6
$\lvert\mathcal{N}(0, 0.33n)\rvert$	0.8	0.07	44	37	84	5.6	18	3.5
$n - 1 - \lvert\mathcal{N}(0, 0.33n)\rvert$	12.8	1.5	39	31	79	8.3	12	4.5

Table 5.34: *mexican hat*: Comparison of different frequency distributions of mutation points over the effective program length n (effmut). $\mathcal{N}(0, 0.33n)$ calculates a normally distributed random number from range $(-n, n)$ with expectation 0 and standard deviation $0.33 \times n$. $\mathcal{U}(n)$ calculates a uniformly distributed integer number within range $[0, n)$. Average results over 100 runs after 1000 generations.

We test the effective mutation operator with two alternative distributions of mutation points over the effective program length. Basically, the selection frequency is either increased towards the beginning of a program (graph sinks) or towards the end of a program (graph root). In doing so, we use a normal distribution $\mathcal{N}(0, 0.33 \times n)$ with expectation

0 and standard deviation $0.33 \times n$. The maximum mutation point n equals the effective program length.

Tables 5.33 to 5.36 compare the performance of the three different distributions, including the uniform distribution $\mathcal{U}(n)$. With all four benchmark problems the performance decreases if the mutation probability is higher at the end of a linear program, and, is almost not affected or even better if this is true at the beginning. These effects directly follow from the functional structure of the genetic programs.

An instruction close to the program end is most likely located high up in the graph structure where the graph width (number of effective registers) is rather small. Mutations are more destructive in this region since more program paths lead through the instruction nodes. Accordingly, mutation effects are more similar in central and lower graph regions where the graph width is constantly wide.

For that reason a higher mutation frequency at the beginning of a linear program may have a more positive influence on the evolutionary search if a larger number of registers is used. Then the functional program structure becomes more tree-like, as we will see in Section 6.1.

Mutation	CE		#Hits	Length			Variations (%)		
Distribution	*mean*	*std.*		*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
$\mathcal{U}(n)$	13.9	0.7	2	77	71	92	1.1	38	1.9
$\vert\mathcal{N}(0, 0.33n)\vert$	13.8	0.9	1	90	81	90	0.9	44	1.7
$n - 1 - \vert\mathcal{N}(0, 0.33n)\vert$	23.9	1.4	0	91	80	88	1.2	43	2.4

Table 5.35: *three chains*: Comparison of different frequency distributions of effective mutation points.

Mutation	CE		#Hits	Length			Variations (%)		
Distribution	*mean*	*std.*		*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
$\mathcal{U}(n)$	8.8	0.4	2	74	69	93	1.7	24	1.7
$\vert\mathcal{N}(0, 0.33n)\vert$	4.5	0.3	10	86	79	91	1.3	33	1.7
$n - 1 - \vert\mathcal{N}(0, 0.33n)\vert$	14.8	0.8	0	79	72	91	1.8	27	1.6

Table 5.36: *spiral*: Comparison of different frequency distributions of mutation points over the effective program length n (effmut). $\mathcal{N}(0, 0.33n)$ calculates a normally distributed random number from range $(-n, n)$ with expectation 0 and standard deviation $0.33 \times n$. $\mathcal{U}(n)$ calculates a uniformly distributed integer number within range $[0, n)$. Average results over 100 runs after 1000 generations.

A second explanation for the above results may be found in the effectiveness degree that decreases approximately linear over the program length in Figure 5.12. A high connectivity of graph nodes reduces the probability that effective subgraphs are disconnected. This influences the effective step size which has been defined for the imperative program code as the number of (preceding) instructions in a program that become effective or noneffective after an instruction mutation (see Section 5.3). One may assume now that the effective step size increases continuously the more the mutation point is located toward the program end, i.e., the graph root. This it true to a certain extent at least, as will be demonstrated in Section 8.7.2.

In Figure 5.13 the effectiveness degree of instructions increases continuously over a run, too. Mostly due to the use of branches the average effectiveness is significantly higher with the *spiral* classification. Note that in Algorithm 3.4 branch instructions inherit the effectiveness of their corresponding conditional operation.

5.12 Summary and Conclusion

In the beginning of this chapter we defined different (structural and semantic) variation effects and step sizes for the linear program representation. Furthermore, properties of variation operators were formulated that we believe are especially desirable for linear GP. A systematic analysis of possible genetic operators was made that is based on these concepts and properties in part. In doing so, different operators were introduced and compared with respect to performance and complexity of the resulting prediction models. Besides, variation-specific parameters were analysed in this chapter. The most important results may be summarized as follows:

(1) Three basic parameters of linear crossover were identified and analysed. Either a restriction of the segment length or the difference in length between the inserted and the deleted segment (size fair crossover) led to a better performance. Interestingly, in both cases the strongest restrictions produced the best results. Instead, it proved to be more deleterious to limit the distance of crossover points.

(2) Unrestricted segment mutations turned out to be at least as powerful as unrestricted recombination and produced less complex solutions. The difference in performance was smaller for two-segment operators than for one-segment operators here. Segment mutations operate even more successfully if the segments are created fully effectively. This results from a further reduction of both noneffective variations and program size. The larger effective step size is partly relaxed here by the smaller program size which indirectly reduces the absolute step size.

(3) In general, best fitness values were obtained by using relatively small variation step sizes on the level of instructions. In particular, a minimization of the absolute mutation step size (to one instruction) in combination with a guaranteed effectiveness of mutations – concerning a change of the structurally effective code – produced the best performance and the smallest solutions. It appears that even minimum changes of program structure and program size (induce semantic step sizes that) are large enough, on average, to escape from local minima (see also Chapter 8).

The performance of these effective instruction mutations gained from an acceleration of code growth by an explicit grow bias. This was not necessarily true for a larger mutation step size. Actually, the effective program length may even shrink by using multiple instruction mutations.

(4) An additional gain in performance (but larger solutions) was only possible by increasing the proportion of neutral instruction mutations on the effective code. This particularly emphasizes the meaning of neutral variations for the evolutionary progress. Without neutral variations the average survival probability of offsprings seems to be too much reduced here to guarantee a continuous improvement and a growth of code (see also Chapter 9). In general, the induction of neutral variations requires information about program semantics by means of multiple fitness evaluations. These extra computational costs cannot be neglected even if the fitness has to be recalculated only after the (structurally) effective code has been altered. Nonetheless, an explicit control of neutrality has been found computationally affordable on the basis of (effective) evaluations.

(5) If only single effective instructions are varied, the existence of structurally noneffective code in programs has not been found absolutely essential for producing high quality solutions in linear GP. The same is true for noneffective variations. That does not mean, however, that structural introns may not contribute to the evolutionary progress (see Section 9.8.5). Moreover, this is definitely not true for all intron code in programs. Since neutral effective variations were highly profitable this must be valid for semantic introns, too, which result at least partly from these variations.

(6) As far as segment variations, like crossover, are concerned the presence of structural introns reduces the effective step size and takes away pressure from the remaining code to grow and to develop semantic introns, which are usually much harder to detect. Without such an implicit parsimony effect the (effective) solution size grows much larger than necessary. To validate this, we removed all structural introns from population individuals after crossover.

(7) Explicit introns provided a more reliable reduction of effective crossover step size than implicit introns because they may not be reactivated. Both a better fitness and a smaller effective size of solutions were achieved depending on the amount of such empty instructions that is seeded into the initial population. Furthermore, implicit introns – including both structural and semantic ones – occured much less in the presence of explicit introns.

For a summary of results concerning the influence of the different genetic operators on the solution size the reader is directed to Section 9.9.1. Moreover, Chapter 9 will discuss several causes for code growth in linear GP. Again neutral variations will play an important role.

Chapter 6

Analysis of Control Parameters

Contents

In the previous chapter parameters have been analysed that are closely related to a variation operator. In this chapter we analyse influences of more general system parameters that are especially relevant to linear genetic programming. In particular, the number of registers, the number of constants, the population size, and the maximum program length are regarded. Additionally, we compare different initialization techniques for linear genetic programs. Test problems are a classification and a regression, *spiral* and *mexican hat*, that have both been introduced already in Section 5.8.1.[1]

6.1 Number of Registers

In linear genetic programming saving local information in registers is an implicit part of the imperative representation. Each operation on registers or constants is combined with an assignment of the result to a register that may again serve as an operand in succeeding instructions. We distinguish additional *calculation registers* from the required minimum number of registers that hold the relevant input information (*input registers*). Both problem definitions expect 2 inputs. For the following considerations we assume that all register are variable.

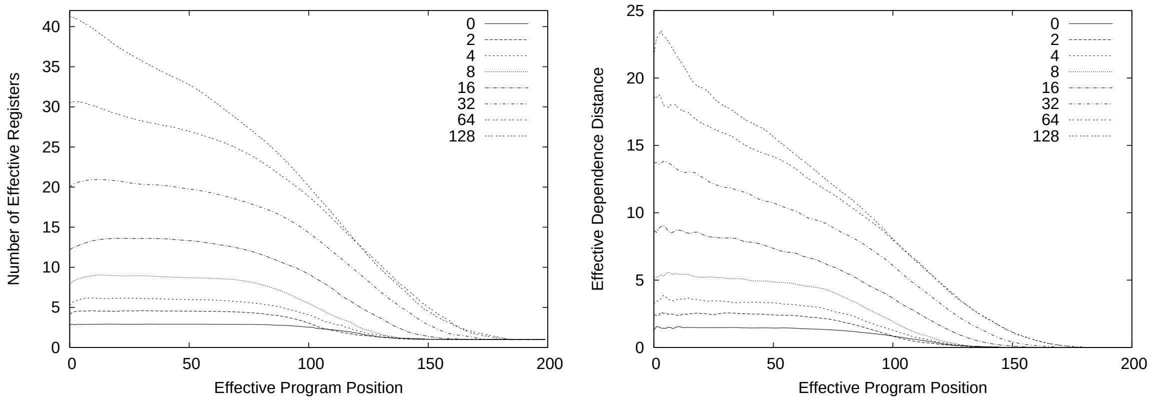

Figure 6.1: *spiral*: Distribution of the effective register number (left) and the effective dependence distance (right) over the (effective) program positions using effective mutations (effmut) with different numbers of calculation registers. Average figures over all programs of the final (1000th) generation and over 100 runs. The standard deviation of program lengths ranges between 5 instructions (0 calculation registers) and 10 instructions (128 calculation registers).

The number of registers is a crucial point for the performance of linear GP. If the number of inputs is low and only a few registers are provided additionally the register contents will be overwritten more often. This makes complex calculations and, thus, the emergence of complex problem solutions more difficult. If too many registers are provided, on the other hand, the search space of possible solutions is unnecessarily blown up. Besides, a lot of programs may be semantically identical in the initial population since the probability is low that instructions manipulate effective registers (see also Section 2.3.1). Hence, there is an optimum number of registers for each problem that represents the best trade-off.

[1]The only difference to the configuration in Section 5.8 is that *mexican hat* is treated with a complete function set $\{+, -, \times, /, x^2, e^x\}$ that allows the optimum solution to be found. However, this happens too rarely to be mentioned here.

#Calculation Registers	CE *mean*	*std.*	#Hits	Length *abs.*	*eff.*	%	Variations (%) *constr.*	*neutral*	*noneff.*
0	24.7	0.5	0	77	73	96	1.8	30	0
2	10.8	0.6	0	82	76	92	1.9	26	0
4	7.6	0.4	2	86	78	91	1.7	25	0
8	6.8	0.3	3	97	86	89	1.4	26	0
16	**6.1**	0.3	3	111	96	86	1.0	30	0
32	8.8	0.4	0	132	110	83	0.6	35	0
64	11.9	0.5	0	144	113	78	0.4	41	0
128	17.2	0.6	0	153	108	70	0.3	49	0

#Calculation Registers	#Effective Registers	Effectiveness Degree	Dependence Distance
0	1.9	5.5	1.4
2	3.4	4.0	2.3
4	4.7	3.3	3.1
8	7.1	2.6	4.5
16	10.8	2.1	6.6
32	15.7	1.7	9.0
64	20.9	1.4	11.2
128	25.1	1.2	12.5

Table 6.1: *spiral*: Effects of different register numbers using effective mutations (effmut2, B1). Number of input registers is 2. Calculation registers are initialized with constant 1. Average results over 100 runs after 1000 generations.

It has to be noted that additional registers may not be beneficial at all for problems that feature a high number of inputs already by definition. Since not all inputs may be relevant for a solution, calculations may not require additional registers for a better performance. In such a case the larger search space would outweigh the advantages.

In this section we investigate how the number of (calculation) registers affects the system behavior. Besides prediction quality, program length and variation effects, the functional structure of *effective* linear programs is analysed, including the number of effective registers, the effectiveness of instructions, and the distance of depending effective instructions (see Section 3.4).

When generating mutations effectively (effmut) good solutions may still be found even with the highest number of registers (see Table 6.1). This is in contrast to free mutation where the prediction error increases significantly beyond a certain register number (in Tables 6.2 and 6.3). Since the effective mutation operator selects the destination register of newly inserted instructions effectively (see Section 5.10.4) the evolutionary process becomes more independent from the total number of registers. That is, the drawback of a larger search space is better counterbalanced.

One can see that for effective mutations the number of effective registers grows with the total number of registers even if the proportion of effective registers decreases. If mutations are generated freely (mut) the probability for selecting an effective register and, thus, the performance decreases directly with the total number of registers. The resulting higher rate of noneffective variations promotes the emergence of more noneffective instructions.

#Calculation Registers	CE *mean*	CE *std.*	Length *abs.*	Length *eff.*	Length %	Variations (%) *constr.*	Variations (%) *neutral*	Variations (%) *noneff.*
0	26.9	0.5	105	59	56	0.6	47	33
2	14.8	0.5	120	63	52	0.4	48	38
4	12.5	0.4	128	66	52	0.3	49	41
8	**10.5**	0.4	136	67	49	0.2	53	45
16	11.8	0.4	145	68	47	0.1	58	50
32	17.2	0.6	148	59	40	0.1	68	61
64	40.4	1.2	142	26	18	0.0	86	82
128	66.5	1.2	135	8	6	0.0	94	93

#Calculation Registers	#Effective Registers	Effectiveness Degree	Dependence Distance
0	1.8	3.5	1.2
2	3.1	2.9	2.1
4	4.4	2.7	2.8
8	6.5	2.3	4.1
16	9.7	1.9	6.0
32	12.7	1.5	7.2
64	9.3	1.0	4.5
128	4.7	0.6	1.7

Table 6.2: *spiral*: Effects of different register numbers using free mutations (mut, B1). Number of input registers is 2. Calculation registers are initialized with constant 1. Average results over 100 runs after 1000 generations.

In general, it may be assumed that the optimum number of provided registers depends on the problem structure as this is true for a sufficient maximum program length. Recall that these parameters determine the size and the shape of the program graph. For both problems tested here the optimum number of calculation registers lies around 8. However, this may be different at least for problem definitions with a higher number of inputs. Beyond a certain number of input registers additional registers may not have a positive influence anymore, but only increase the search space of solutions.

As we know from Chapter 3.3 the number of effective registers corresponds to the width of the (effective) program graph. The more registers are available the wider these graphs may become. Concurrently, the connection degree of graph nodes, or, more precisely, the number of incoming edges per node (indegree) decreases with higher register numbers. A constant indegree of 1 means that the graph represents a tree program. Recall that the connectivity of nodes corresponds to the effectiveness degree of instructions in the imperative representation. The effectiveness degree provides information about how often the result of an effective instruction is used by other program instructions. Figure 6.1 shows the average distribution of the number of effective registers over the effective program positions. Obviously, the functional structure becomes more and more tree-shaped with a higher number of registers if we take into consideration that the average effectiveness degree over all program instructions in Table 6.1 converges to 1. When using free mutations with many registers this value may even be smaller than 1 (see Tables 6.2 and 6.3). In this case, the rate of effective instructions is so low, on average, that many programs do not even hold a single effective instruction, i.e., have effective length 0. At least, for a

#Calculation Registers	SSE		Length			Variations (%)		
	mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
0	7.6	0.9	52	37	71	3.2	29	26
2	6.0	0.9	66	39	59	1.7	41	39
4	3.0	0.5	73	39	53	1.1	49	47
8	**1.3**	0.2	80	35	44	0.6	59	58
16	3.6	0.5	78	25	32	0.2	73	72
32	21.1	1.0	68	12	18	0.0	86	85
64	42.1	1.2	61	5	8	0.0	92	91

#Calculation Registers	#Effective Registers	Effectiveness Degree	Dependence Distance
0	1.7	1.4	1.1
2	2.7	1.3	1.6
4	3.3	1.3	2.0
8	4.3	1.2	2.5
16	4.4	1.1	2.3
32	3.4	0.9	1.6
64	2.5	0.7	1.0

Table 6.3: *mexican hat*: Effects of different register numbers using effective mutations (mut, B0). Number of input registers is 2. Calculation registers are initialized with constant 1. Average results over 100 runs after 1000 generations.

mutation-based program induction we may conclude that a tree structure does not always represent the optimum functional shape for a genetic program.

The number of registers may also influence the length of programs in linear GP. When using effective mutations the effective size grows continuously with the register number (see Tables 6.1). Larger program graphs are required to represent the same solution if subgraphs are hardly connected (used) more than once. As a result, the functional representations grow in width and more (effective) instructions are needed for the imperative representation. This, in turn, increases the average distance between two depending instructions in the effective program (see Figure 6.1). Recall that two depending instructions correspond to two directly connected instruction nodes in the graph representation.

It has to be mentioned here, however, that the program length is not always affected by the register number when using explicitly effective mutations (effmut). For the *mexican hat* problem we found no significant change in the amounts of effective and noneffective code, not even with very many registers (not shown). Nevertheless, the same principle developments may be observed in terms of the structural program analysis.

Similar developments may be observed with free mutations only until a certain maximum number of registers. Beyond that point the complexity of solutions – including the size and proportion of effective code, the average number of effective registers, and the average effective dependence distance – decrease again (see Tables 6.2 and 6.3).

6.1.1 Initialization of Registers

The results in Table 6.1 are obtained by using only as many input registers as there are input values. The remaining registers are initialized with a constant (1 here). We have seen that above a certain register number the performance decreases again. At this point, the probability for selecting an input register becomes too low. This problem can be overcome by initializing more registers with input values. As a side-effect, input values get lost less likely through overwriting in a calculation.

In the following experiments, we assign an input value to each register such that for each input about the same number of registers is used. As one can see in Table 6.4, the average prediction error stays more-or-less the same even if the optimum number of registers is exceeded. Apparently, the larger search space by more registers is better counterbalanced here than with the constant initialization in Table 6.1. Moreover, the prediction error has been found twice as small while the hit rate is significantly higher.

#Calculation	CE		#Hits	Length			Variations (%)		
Registers	*mean*	*std.*		*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
0	24.7	0.5	0	77	73	96	1.8	30	0
2	9.5	0.4	1	82	76	92	1.9	25	0
4	5.5	0.3	3	84	76	91	1.8	24	0
8	3.4	0.3	16	91	80	88	1.6	25	0
16	**3.0**	0.2	9	103	89	86	1.3	26	0
32	3.4	0.3	15	113	95	84	1.0	29	0
64	3.6	0.3	11	126	102	81	0.9	32	0
128	3.9	0.3	7	133	103	77	0.7	34	0

Table 6.4: *spiral*: Effects of different register numbers using effective mutations (effmut2, B1). Number of input registers is 2. Calculation registers are initialized with input values. Average results over 100 runs after 1000 generations.

The average number of effective registers has been found quite similar as for the standard initialization (undocumented). That means calculations do not involve a larger number of effective registers only because all registers are initialized with input data. This is also reflected by similar (effective) solution sizes.

Nevertheless, input values may be *used* more frequently in a genetic program if held in more than one register. Otherwise, each input value may be read out (used as an operand) only until its register is overwritten for the first time. As indicated in Section 3.3, such operand registers label variable sink nodes (terminals) in the functional representation. More input registers mean more variable terminals.

The above behavior has only been observed with the *spiral* classification when using effective instruction mutations. With the *mexican hat* approximation the performance improves only slightly and gets worse again for higher register numbers just as if calculation registers were initialized constantly.

Tables 6.5 and 6.6 show how crossover results are influenced by the number of registers if each register holds an input value. First, in both test cases the average prediction error improves to a certain extent by using more calculation registers. Second, especially the *mexican hat* task is much better solved with crossover if all registers are initialized with inputs, compared to a constant initialization of 4 calculation registers (see, e.g., baseline results with maximum length 200 in Tables 6.14 and 6.13).

A lower proportion of effective code, i.e., a higher ratio of noneffective code (structural introns), may be maintained by more registers. This results mostly from the fact that a smaller *proportion* of registers is effective, on average, and is illustrated in Figure 6.2 for

#Calculation Registers	SSE mean	SSE std.	Length abs.	Length eff.	Length %	Variations (%) constr.	Variations (%) neutral	noneff.
0	11.4	0.9	144	71	49	6.4	21	14
2	5.9	0.8	167	65	39	5.3	24	19
4	2.8	0.5	177	59	33	4.6	27	23
8	**1.7**	0.2	184	52	28	3.8	30	26
16	1.7	0.2	187	43	23	3.1	34	31
32	4.5	0.4	186	34	18	2.6	45	41
64	10.2	1.3	187	25	13	1.8	51	49

Table 6.5: *mexican hat*: Effects of different register numbers using crossover (cross). Number of input registers is 2. Calculation registers are initialized with input values. Average results over 100 runs after 1000 generations.

linear crossover. If only few or no additional registers are provided the effective length depends more strongly on the absolute length. It is interesting to note that in this case the absolute length grows smaller while the effective length grows larger. In Table 6.6, instead, almost only the effective code is altered while the absolute length is more-or-less the same for all register configurations due to both a faster code growth and the maximum length bound.

#Calculation Registers	CE mean	CE std.	Length abs.	Length eff.	Length %	Variations (%) constr.	Variations (%) neutral	noneff.
2	23.8	0.7	186	109	58	3.5	24	13
4	19.0	0.6	187	102	55	3.2	24	15
8	15.3	0.5	187	101	54	2.8	23	15
16	**13.0**	0.4	190	98	52	2.2	23	15
32	15.1	0.5	192	87	45	1.8	25	17
64	18.2	0.5	192	77	40	1.5	30	20
128	22.7	0.5	192	67	35	1.2	35	24

Table 6.6: *spiral*: Effects of different register numbers using crossover (cross). Number of input registers is 2. Calculation registers are initialized with input values. Average results over 100 runs after 1000 generations.

The smaller ratio of effective code is correlated with a higher number of noneffective variations. Due to a large absolute step size the rate of noneffective operations is increased less with the number of registers than this has been found for (free) instruction mutations in the last section. Nevertheless, this rate still increases almost by 35 percent for *mexican hat*. As noted before, a smaller proportion of effective code reduces the effective step size of segment variations like crossover. In doing so, performance may be improved until the point where the rate of effective operations or the effective code is reduced too much to produce good solutions.

We will demonstrate in Section 8.7.2 that the register number has a negative influence on the effective step size, too, that is independent from the applied variation operator. A decreasing effectiveness degree of instructions makes larger deactivations of code more likely. Therefore, the effective code becomes more brittle if more registers are available.

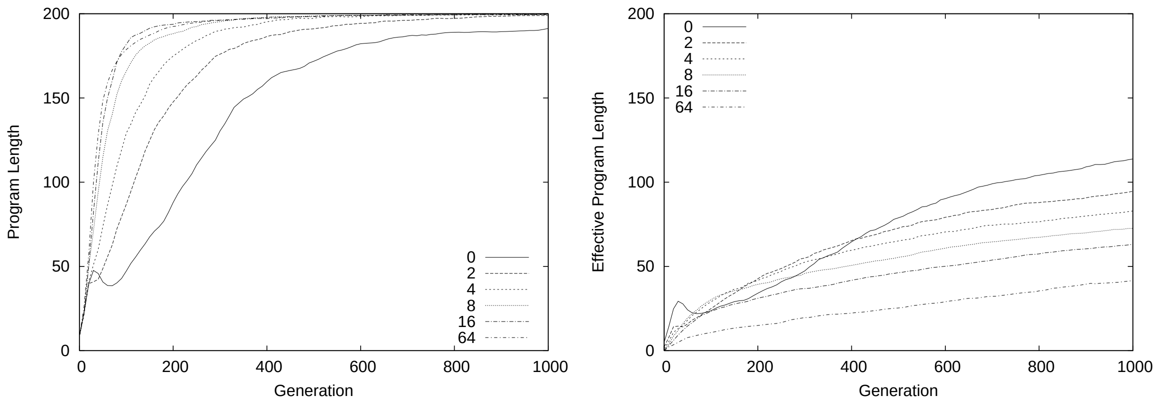

Figure 6.2: *mexican hat*: Development of absolute program length (left) and effective program length (right) for different numbers of calculation registers using crossover (cross). Absolute length grows faster with more registers while effective length grows more slowly. That is more intron code occurs with more registers. Average figures over 100 runs.

6.1.2 Constant Registers

Another probability to protect the input information – besides using multiple input registers – is not to use input registers as a destination register in instructions. Constant (input) registers are different from registers being initialized with a constant such that their initial contents (input data) may change with each fitness case but may not be overwritten during program execution. As a consequence, program output(s) have to be stored in registers that are different from such input registers. In the graph interpretation of linear programs constant input registers denote constant sinks that may become successor of any node.

This technique has not been found to produce better results than those are obtained with the standard configuration, at least for the two problems under investigation. Note that in contrast to the approach from Section 6.1.1 the probability for selecting an input register decreases here with the total number of registers. Moreover, if all input registers are constant, a higher number of variable registers has to be provided additionally. This increases the search space of programs.

6.2 Number of Output Registers

Usually one register is explicitly designated in linear GP for holding the final output of a program after execution. This may be any writable register, including input registers (see Section 2.1.1). Let us assume for the following considerations that there is only one output defined for a problem. In the normal case, the output register is static, i.e., all programs save their result in the same register. Alternatively, we propose here to change the output register of programs dynamically during a run.

If the fitness of an individual program is calculated it is executed once for each fitness case. After each program execution the contents of all registers may be saved. Then the program fitness can be calculated efficiently multiple times without further executions while each time the contents of another register (from a predefined subset) may be used as the program output. The output register which a program performs best with during training is saved and may not be changed anymore when the program is applied to unknown data, e.g., when its generalization performance is tested.

In doing so, it is important that the output register is fixed for all fitness cases. At least, it is not feasible to change this register in a non-deterministic way. For instance, we may not simply select the output register whose value is closest to the desired output of a fitness case. This would make a "prediction" possible only if the correct output is already known in advance. The resulting GP models would be incapable to decide on unknown data in a deterministic way.

Each (output) register or, more precisely, the last instruction in a program that manipulates it, labels the root of one contiguous component in the functional interpretation (see Section 3.3). If the output register is static only one contiguous component of a program graph is effective and tested. If the output register is dynamic a more-or-less different component becomes effective for each designated output register.

On the imperative level the distinction between effective and noneffective instructions in a linear program depends on the definition of the output register. But even if each register may hold the output there may be still many instructions left that are noneffective for *all* output registers.

We only note here that results were disappointing. For both test problems, *mexican hat* and *spiral*, the performance has not been found better with a dynamic output register than with a static output register. While crossover (cross) results were approximately identical in both cases, solution finding with instruction mutations (mut) was much more limited when using the final contents of all 6 registers (see Section 5.8). In general, the best output register (saved with the best individual) has been found to change mostly at the beginning of a run. After a while one output register (graph component) is winning out dominating. This does not only show that the output register is better fixed, but encourages the exclusive development of a single graph component, too, as done by the effective mutation operator.

6.3 Rate of Constants

Besides instruction operators and registers, constants represent the third basic component of a linear genetic program. The reader may recall from Section 2.1.2 that we allow only one of two operands of an instruction to hold a constant. First, assignments of constant values are avoided explicitly, in this way, e.g., $r_0 := 1 + 2$ or $r_0 := sin(1)$. Second, there is at least one register for each program position whose manipulation may influence the effective code. Otherwise, the number of effective registers may become zero and effective variations would not be possible at each program position. As a result, the potential for creating structural noneffective code (see Section 3.2) is increased even if this does not mean that the rate of noneffective code really becomes larger.

The same arguments hold for constant register operands, too, that have been discussed in Section 6.1.2.[2] While the number and the range of constants ($\{0, .., 9\}$ here) in the terminal set are rather problem-dependent parameters we investigate the number of operands in linear genetic programs that represent a constant value. This is the number of instructions that hold a constant at all and depends on the probability for which constants are created during mutation or during the initialization of programs. As a standard configuration a

[2]In our implementation constant values are saved in registers (see Section 2.1.1). Instead of holding constants directly in instructions they are addressed over register indices. These "registers" differ from what is referred to as a constant (input) register here such that their value may not change between different executions of a program.

Constants (%)	SSE		Length			#Eff.	Eff.	Depend.
	mean	*std.*	*abs.*	*eff.*	%	Registers	Degree	Distance
0	1.2	0.2	41	36	88	3.7	1.5	2.0
50	0.6	0.06	33	28	85	2.8	1.2	1.6
100	33.8	0.01	18	11	60	1.0	0.9	0.9

Table 6.7: *mexican hat*: Effects of different rates of instructions holding a constant (effmut, B1). Average results over 100 runs after 1000 generations.

Constants (%)	CE		Length			#Eff.	Eff.	Depend.
	mean	*std.*	*abs.*	*eff.*	%	Registers	Degree	Distance
0	10.1	0.5	62	59	96	5.0	3.7	2.7
50	8.4	0.4	66	62	95	4.6	3.3	3.1
100	12.8	0.5	69	63	91	4.1	2.5	3.9

Table 6.8: *spiral*: Effects of different rates of instructions holding a constant (effmut, B1). Average results over 100 runs after 1000 generations.

probability of 50 percent has been used in most experiments. In general, this has been found to be a good choice. Note that the composition of programs, i.e., the proportion of a program element, in the population is strongly influenced by selection, too.

Tables 6.7 and 6.8 compare prediction performance, program size and (functional) program characteristics for different rates of constants. Interestingly, the prediction error increases less if constants are not used at all rather than if almost each instruction holds a constant. Especially for the *mexican hat* problem the performance drops drastically in the latter case. Moreover, the (effective) program size becomes smaller the more instructions hold a single register operand only. Both may be explained if we have a look at the functional structure of such programs. If all instructions use the result of only one other instruction the graph is reduced to a linear list of operator nodes. Such a restriction makes the emergence of successful complex solutions impossible. As a result, the average number of effective registers, the average degree of effectiveness and the average effective dependence distance are constantly 1 for all (effective) programs. In Table 6.7 the last two parameters are slightly smaller due to programs with effective length 0. The first parameter calculates 1 for these programs because at least the output register stays effective.

The results in Table 6.8 show, by contrast, that the *spiral* classification is much less influenced by the rate of constants in linear programs. This is true for almost all observed features. The reason is that branches are used with this problem. Then the data flow is not restricted to a linear list of nodes even if all instructions operate on a single register.

6.4 Population Size

The evolutionary algorithm that is used throughout this thesis (see Section 2.3.2) operates with a steady-state population and tournament selection. The population size is an important parameter when comparing mutation-based with recombination-based variation. We apply either instruction mutations or linear crossover for macro variations.

The performance of recombination, by definition, depends more strongly on the composition of the genetic material in the population. Even if the building block hypothesis is not valid (see discussion in Section 5.7.7), the genotype diversity of a population influences the innovation positively that may be introduced by the recombination operator. Larger populations allow a higher diversity than smaller ones.

Instead, the population size may be supposed to have a lower influence on the performance of mutations which introduce new genetic material regularly into the population. It has to be noted, however, that diversity is not the only system attribute that is influenced by the population size. Even a pure mutation-based approach may profit from the higher parallelism of search points in larger populations. Moreover, the population size influences the complexity of solutions (see below).

Population	Generations	CE		Length			Variations (%)		
Size	#	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
10	100000	45.3	3.6	109	83	76	2.9	41	31
100	10000	23.5	0.7	196	125	64	3.8	18	11
1000	1000	26.1	0.7	185	102	55	3.6	23	14
10000	100	24.7	0.4	125	53	42	3.0	38	23

Table 6.9: *spiral*: Effects of population size on crossover (cross). Average results over 100 runs after 1000000 evaluations.

Population	Generations	SSE		Length			Variations (%)		
Size	#	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
10	100000	23.2	2.4	143	74	52	4.5	39	24
100	10000	12.4	1.4	196	91	46	5.6	24	18
1000	1000	16.1	1.5	180	60	33	4.5	28	25
10000	100	11.9	1.3	97	21	22	4.4	36	33

Table 6.10: *mexican hat*: Effects of population size on crossover (cross). Average results over 100 runs after 1000000 evaluations.

If the solution quality is compared for different population sizes on the basis of generations, bigger populations always produced better results (not shown). This is true because more evaluations are performed per generation while the average number of evaluations (and variations) per individual (position in the population) remains constant. The number of evaluations equals the number of variations if only newly created individuals are evaluated. Only measuring runtime on the basis of fitness evaluations guarantees a fair comparison. Comparing evaluations after effective variations (effective evaluations, see Section 5.2) is not necessary since the proportion of effective variations is not influenced significantly by the population size.

The smaller the population size is set the more often an individual (position) is selected for variation and the more generations happen within a certain period of evaluations. Code diversity may be lower not only because of less individuals but because the same individual index is reproduced more frequently (by overwriting worse individuals in the steady-state population).

The larger a population is, on the other hand, the more solutions may be developed in parallel. If a population contains too many individuals in relation to the observed number of evaluations the number of variations per individual will not be sufficient to develop successful solutions. Success depends too much on random events then instead of evolutionary progress.

Population Size	Generations #	CE mean	CE std.	Length abs.	Length eff.	Length %	Variations (%) constr.	Variations (%) neutral	Variations (%) noneff.
10	100000	**5.7**	0.3	122	105	86	1.5	16	0
100	10000	7.5	0.4	96	88	92	1.4	29	0
1000	1000	11.6	0.4	51	47	92	2.2	20	0
10000	100	25.5	0.5	24	18	76	3.4	31	0

Table 6.11: *spiral*: Effects of population size on effective mutations (effmut2, B0). Average results over 100 runs after 1000000 evaluations.

Population Size	Generations #	SSE mean	SSE std.	Length abs.	Length eff.	Length %	Variations (%) constr.	Variations (%) neutral	Variations (%) noneff.
10	100000	1.8	0.3	119	66	56	8.2	5.2	0
100	10000	1.1	0.1	70	43	62	9.0	6.3	0
1000	1000	**0.7**	0.05	39	25	64	8.4	9.9	0
10000	100	2.8	0.2	21	12	55	9.0	16.6	0

Table 6.12: *mexican hat*: Effects of population size on effective mutations (effmut2, B0). Average results over 100 runs after 10000000 evaluations.

Tables 6.9 to 6.10 show for both test problems that crossover performs worst in the smallest population (10 individuals here). It is interesting to see that the relative differences in performance are rather low with larger population sizes after the same number of evaluations. When using effective mutations the situation is less clear. For the *spiral* problem best solutions are obtained with the smallest population size (see Table 6.11). The *mexican hat* problem, instead, is solved most successfully with a medium population size (see Table 6.12). Only if the number of generations falls below a certain minimum, the performance decreases again. This example shows that a pure mutation-based approach does not automatically perform better with a smaller population size. In general, crossover performance seems to depend less on the relation of population size and generation number than the performance of instruction mutations.

The different optimum population sizes found for the two test problems may result from a different correlation between solution quality and solution size, too. The population size clearly influences code growth, especially when using effective mutations. But why do programs become larger in smaller population ? On the one hand, an individual may grow larger in a smaller population because it is selected and varied more frequently. This is true as long as larger solutions show a better fitness. Likewise, other causes of code growth than fitness may be reinforced. In particular, more neutral variations *per individual* may create more neutral code (see Chapter 9). This is true even if Tables 6.11 and 6.12 show that the *proportion* of neutral variations per generation is smaller in smaller populations. Note that effective mutations are configured without an explicit grow bias here that would be reinforced otherwise.

On the other hand, especially the small absolute step size of instruction mutations lets programs grow only insufficiently in larger populations since not enough variations and evaluations happen per individual. Large absolute step sizes, instead, such as those are induced by the use of crossover, allow effective programs to be developed more independently from the average effective length in the previous generation (see also Section 6.6).

In most experiments of Chapters 5 and 6 we decided for population size 1000 (and 1000 generations) because this has been found to be a good trade-off between a sufficient training time and a low influence on program growth, especially for instruction mutations.

6.5 Maximum Program Length

The simplest form of growth control in genetic programming is to keep the maximum size limit of programs as small as necessary for representing successful solutions. In linear GP this is the maximum number of program instructions. In the following the influence of the maximum program length is analysed for unlimited linear crossover. In contrast to crossover, effective mutations control the complexity of programs already implicitly. In this case, the upper bound may be just chosen sufficiently large such that it is never reached within the observed period of generations.

Maximum Length	CE *mean*	CE *std.*	Length *abs.*	Length *eff.*	Length %	Variations (%) *constr.*	Variations (%) *neutral*	*noneff.*
20	37.7	0.7	20	16	78	4.5	19	13
50	30.2	0.8	49	34	69	3.9	20	14
100	27.9	0.7	96	59	61	3.8	22	15
200	26.1	0.7	185	102	55	3.6	23	14
500	23.3	0.7	446	216	48	3.5	26	16
1000	**21.7**	0.6	858	392	46	3.3	27	16

Table 6.13: *spiral*: Effects of maximum program length on crossover (cross). Average results over 100 runs after 1000 generations.

Maximum Length	SSE *mean*	SSE *std.*	Length *abs.*	Length *eff.*	Length %	Variations (%) *constr.*	Variations (%) *neutral*	*noneff.*
25	10.3	1.2	25	15	62	5.6	24	22
50	**4.8**	0.8	48	26	54	5.3	24	22
100	8.4	1.2	94	40	43	5.0	26	23
200	16.1	1.5	180	60	33	4.5	28	25
500	20.4	1.5	410	97	24	4.1	32	28
1000	21.0	1.5	751	145	19	3.9	35	31

Table 6.14: *mexican hat*: Effects of maximum program length on crossover (cross). Average results over 100 runs after 1000 generations.

Tables 6.13 and 6.14 show exactly the opposite effect on the performance for the two test problems. While *mexican hat* profits from a smaller maximum size of solutions, *spiral* does not. Most successful solutions for the regression problem may be assumed in lower dimensional regions of the search space, while for the classification task even very large (effective) solutions still perform better since these allow a higher specialization by intergrating more branches. In other words, fitness is positively correlated to program size for the latter problem. For the former problem this is so until a sufficient maximum size only. Beyond that correlation becomes rather negative.

One important general conclusion from the fact that even very long linear programs still improve results, is that their functional representation is not restricted in scalability. This is true for the depth of the directed graph as well as for the graph width. As argued in Section 6.1 the imperative programs may even represent large trees if the number of registers is sufficiently high.

In both test cases, the average absolute length and the average effective length per run increase with the maximum bound. Concurrently, the proportion of effective code decreases. The average length in the population (see Figure 6.3) converges quickly to the maximum during a run depending on how large the maximum is configured. This developement is characterized by an explosive increase of program lengths in early generations. One reason for this is the unlimited exchange of instruction segments during crossover. Another reason is the noneffective code that may grow almost without restrictions since the program fitness is not directly influenced by it. The proportion of structural introns is lower in Table 6.13 than in Table 6.14 due to the higher tendency for larger effective solutions.

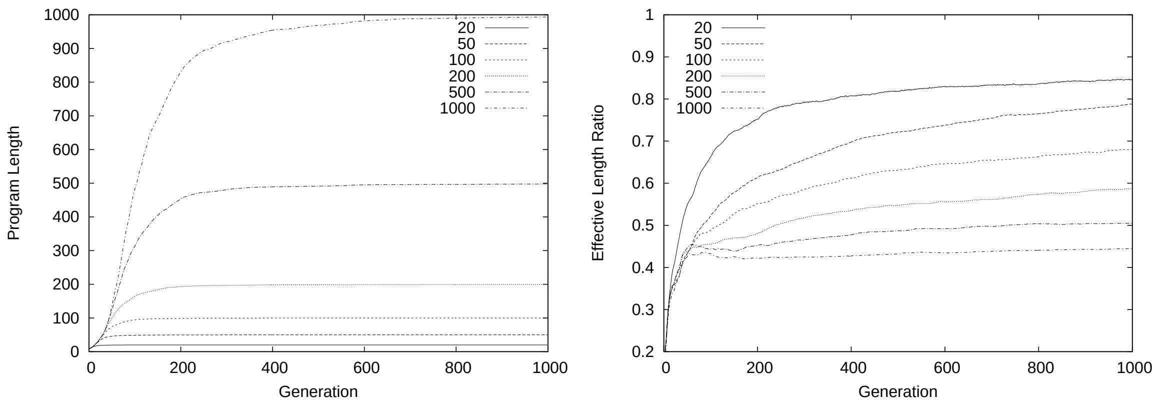

Figure 6.3: *spiral*: Development of absolute program length (left) and relative effective length (right) for different maximum bounds using crossover (cross). The less code growth is restricted by the maximum complexity bound the less the proportion of effective code increases over a run. Average figures over 100 runs.

Figure 6.3 shows exemplarily the development of both the absolute program length and the proportion of effective code (*relative effective length*) over a run for different maximum lengths. In general, the larger programs may grow the later the maximum is reached during a run. Interestingly, the proportion of effective code remains mostly constant over a run here for the highest maximum bound. Otherwise, the rate of effective code increases because this type of code may still grow even if the absolute length of a program is already maximum. In doing so, noneffective code is replaced by effective code during a run.

In general, a higher proportion of noneffective instructions in programs reduces the average effective step size of crossover. That means, by Definition 5.4, that the average *amount* of effective instructions decreases that is exchanged during a crossover operation. However, this is valid only for the same program length, not if the the maximum complexity bound is extended. First, the effective code increases indirectly together with the absolute length. In general, it is more difficult to maintain a small amount (not proportion) of effective code in a large program context. A larger amount of noneffective code implies a larger amount of effective code, especially if only a few registers are used (see above). Second, the absolute step size of unrestricted crossover, i.e., the total amount of exchanged instructions, grows proportionally to the program size. For both reasons, the effective crossover step size *increases* here. Nonetheless, it may be argued that the increasing intron rate in Tables 6.13 and 6.14 results at least partly from a higher need for protection against increasing absolute step sizes (see also Chapter 11).

For the same reasons the proportion of noneffective variations increases only slightly compared to the proportion of noneffective code. We only note here that this is different with free instruction mutations whose minimum step size lets the effectiveness of operations more directly depend on the proportion of effective instructions.

The reader may note also that code growth is influenced by a larger maximum size bound only to a certain extent when using segment mutations even if the segment length is not explicitly restricted (see Section 9.9.2).

Finally, the influence of the (maximum) program length on the number of effective registers (graph width) and the degree of effectiveness (connectivity of nodes) has been found quite low (undocumented).

6.6 Initialization of Linear Programs

The initialization of individuals denotes the first step of an evolutionary algorithm. In genetic programming it determines the size, shape and diversity of programs in the initial population. Depending on the type of program representation different strategies may be developed. Popular method for initializing tree populations will be introduced in Section 7.1.2. In this section we define and compare different initialization method for the linear representation. Basically, the following forms are discerned:

- ☐ During *free initialization* programs are filled with instruction that are created randomly.
- ☐ A *(fully) effective initialization* builds programs completely from effective code starting with the last instruction (see Section 5.10.4).
- ☐ *Maximum initialization*: The absolute length of all initial programs equals the maximum program length.
- ☐ *Constant-length initialization*: All programs have the same initial length.
- ☐ *Variable-length initialization*: Initial program lengths are selected uniformly distributed from a predefined range.

All strategies, except for the effective initialization, apply to the absolute length of programs. The initial effective length may vary freely and increases automatically with the initial absolute length. In contrast to a free initialization with longer absolute programs, a fully effective initialization allows a higher (effective) diversity of initial programs without increasing the total amount of genetic material.

If programs are initialized too long, on average, they may be more inflexible during evolutionary manipulations. This is especially true if the average step size of macro variations is rather small. The minimum step size of instruction mutations lets the best prediction quality be achieved with rather small initial lengths (see Tables 6.15 and 6.17). Moreover, both the absolute size and the effective size of solutions increase clearly by effective mutations if a longer initial size is chosen. In general, it seems to be more difficult for the evolutionary algorithm to follow a search path from a complex region of the search space to another complex region (with better programs) than to start with low-complex programs.

Figure 6.5 shows, exemplarily for the *mexican hat* problem, how the program length develops by applying effective mutations without an explicit grow bias. Different (effective)

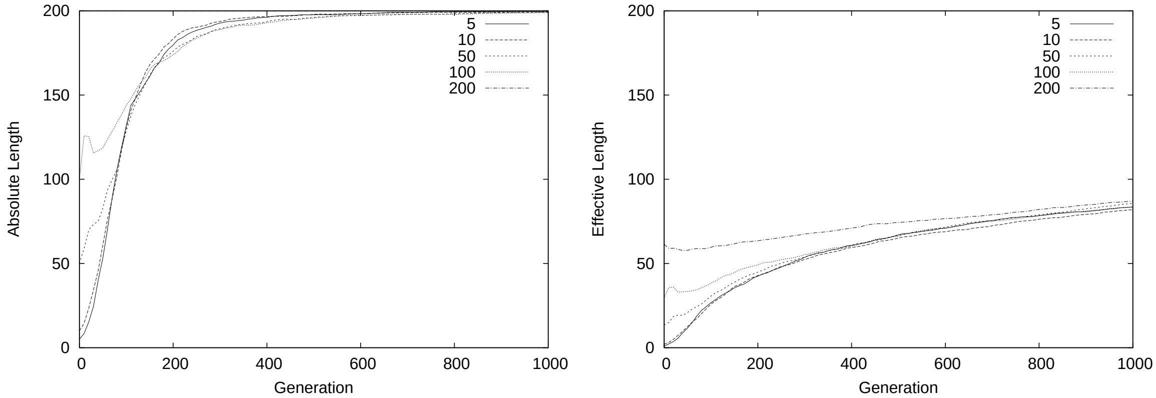

Figure 6.4: *mexican hat*: Development of absolute program length (left) and effective program length (right) for different inital lengths using free initialization and crossover (cross). Average figures over 100 runs.

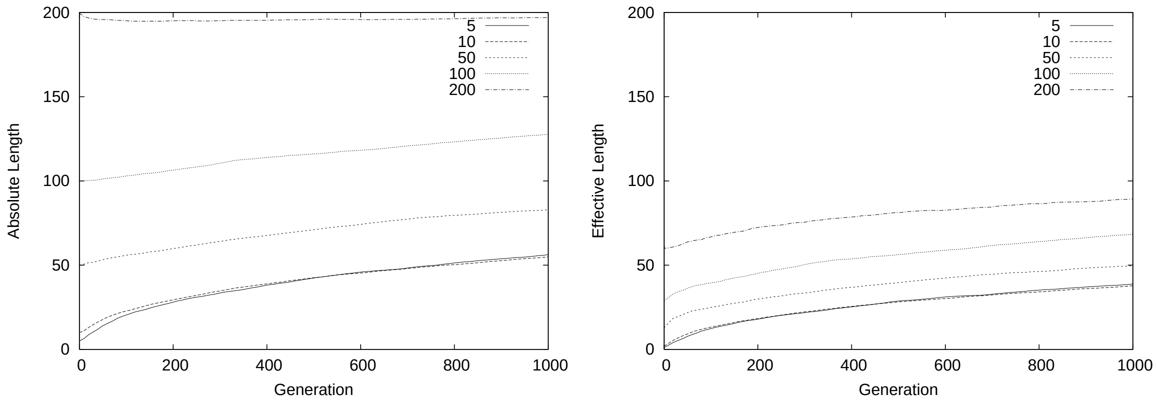

Figure 6.5: *mexican hat*: Development of absolute program length (left) and effective program length (right) for different initial lengths using free initialization and effective mutations (effmut2, B0). Average figures over 100 runs.

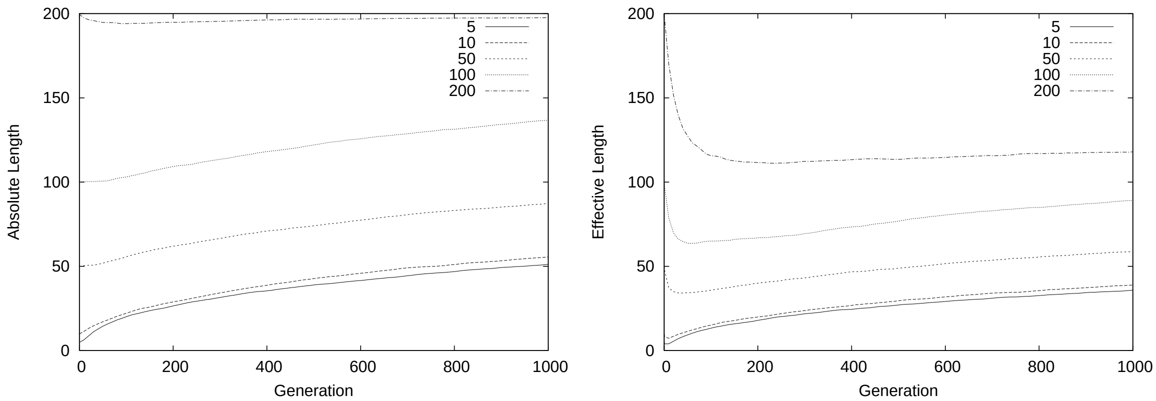

Figure 6.6: *mexican hat*: Development of absolute program length (left) and effective program length (right) for different initial lengths using fully effective initialization and effective mutations (effmut2, B0). Average figures over 100 runs. (Similar figures found for the *spiral* problem.)

Initial Length	SSE *mean*	SSE *std.*	Length *abs.*	Length *eff.*	Length %	Variations (%) *constr.*	Variations (%) *neutral*	*noneff.*
5	**0.6**	0.06	39	26	67	8.3	10	0
10	0.7	0.1	39	26	65	8.5	10	0
50	0.9	0.1	70	38	54	8.7	9	0
100	1.2	0.1	115	54	47	8.6	9	0
200	3.5	0.4	196	79	40	8.6	11	0
1–200	1.7	0.1	130	58	45	8.5	10.5	0.0

Table 6.15: *mexican hat*: Effects of initial program length on effective mutations (effmut2, B0) using free initialization. Maximum program length is 200. Average results over 100 runs after 1000 generations.

Initial Length	SSE *mean*	SSE *std.*	Length *abs.*	Length *eff.*	Length %	Variations (%) *constr.*	Variations (%) *neutral*	*noneff.*
5	0.6	0.06	36	25	69	8.5	10	0
10	**0.4**	0.05	40	28	69	8.6	9	0
50	1.0	0.1	72	48	67	8.6	9	0
100	2.5	0.2	120	77	64	8.4	11	0
200	6.0	0.5	196	118	60	7.7	16	0
1–200	1.5	0.2	119	59	50	8.2	11	0

Table 6.16: *mexican hat*: Effects of initial program length on effective mutations (effmut2, B0) using effective initialization. Maximum program length is 200. Average results over 100 runs after 1000 generations.

initial lengths are continuously increased during a run for almost the same amount. Thus, it strongly depends on the initialization here how large programs may become during a certain period of generations. Apparently, maximum mutation steps of one instruction are too small to break up larger initial structures sufficiently.

In Figure 6.4 we can see, by comparison, that the more (effective) code exists initially the less the (effective) length grows in the course of the evolutionary algorithm when using (unrestricted) crossover. Interestingly, the effective size converges to almost the same value in the final generation, no matter how large the initial programs are constituted. Similar results have been observed with (unrestricted) segment mutations. Apparently, larger step sizes allow (effective) programs to grow almost independently from their initial (effective) size.

Neither with crossover nor with effective mutations the average effective length falls below its initial level after a free initialization. Instead, a more-or-less rapid drop of effective length occurs at the beginning of runs if longer individuals are initialized fully effectively (see Figure 6.6). This has been found with both benchmark problems here. The decrease in average effective length results from early deactivations. Nevertheless, the absence of inactive code in the initial population reduces the emergence of this code during a run. As a replacement, the effective code develops larger than with the standard initialization. This is also the reason why an effective initialization results in a worse performance for larger initial programs (compare Table 6.15 with Table 6.16 and Table 6.17 with Table 6.18). A slightly better performance is obtained, however, with smaller initial lengths probably due to a higher diversity of the initial effective solutions.

Initial	CE		Length			Variations (%)		
Length	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
5	**10.1**	0.5	50	46	92	2.1	20	0
10	11.3	0.5	55	50	91	2.1	20	0
50	14.2	0.6	82	69	85	1.9	21	0
100	16.8	0.6	128	100	78	1.7	24	0
200	22.3	0.6	197	136	69	2.0	23	0
1–200	16.1	0.5	113	90	79	1.8	23	0

Table 6.17: *spiral*: Effects of initial program length on effective mutations (effmut2, B0) using free initialization. Maximum program length is 200. Average results over 100 runs after 1000 generations.

Initial	CE		Length			Variations (%)		
Length	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
5	10.0	0.5	50	46	92	2.2	20	0
10	**8.6**	0.5	53	48	91	2.1	21	0
50	16.4	0.6	83	74	89	2.2	21	0
100	22.7	0.6	132	116	87	2.2	22	0
200	31.0	0.5	198	175	88	3.4	19	0
1–200	16.0	0.7	88	79	89	2.2	21	0

Table 6.18: *spiral*: Effects of initial program length on effective mutations (effmut2, B0) using effective initialization. Maximum program length is 200. Average results over 100 runs after 1000 generations.

If the initial lengths are too small, many programs may be identical in both (effective) structure and semantics. In particular, many initial programs may have effective length zero. Initialization influences diversity in such a way that both *more* or *longer* programs allow a higher diversity. If variation is dominated by recombination the composition of the initial population has a stronger influence on the success of solutions (see also Section 6.4). This is another reason, besides its larger absolute step sizes, why crossover may perform better with a higher amount of initial genetic material. At least the *mexican hat* problem is better treated with longer initial programs in Table 6.19. If variation is based primarily on mutations, instead, the initial diversity is less important since new material is seeded regularly into the population anyway.

We have seen in Section 5.9.5 that smaller effective lengths may be maintained in linear programs by building the initial programs partly from empty instructions (explicit introns). Additionally, the proportion of implicit introns is significantly reduced in this way wherefore reactivations are much less likely. For both reasons, crossover steps become smaller in terms of the effective code. This may not be achieved already by increasing the initial program length as demonstrated in Figure 6.4. In generation 0 the proportion of effective code is more-or-less the same for different absolute lengths. If the program size is doubled the effective size doubles, too. Consequently, both the absolute step size and the effective step size of unrestricted linear crossover increase with the initial program length.

Because the rate of noneffective code is more-or-less unchanged the same is true for the probability of a crossover operation to become noneffective (see Tables 6.19 and 6.20). In general, the probabilities for neutral variations and constructive variations are not affected

Initial Length	SSE mean	SSE std.	Length abs.	Length eff.	Length %	Variations (%) constr.	Variations (%) neutral	Variations (%) noneff.
5	15.0	1.5	179	60	33	4.7	29	25
10	15.5	1.4	180	58	32	4.4	29	25
50	7.4	1.0	180	61	34	4.8	26	23
100	**5.4**	0.6	184	63	34	5.1	25	21
200	6.9	0.6	200	73	37	5.3	25	19

Table 6.19: *mexican hat*: Effects of initial program length on crossover (cross) using free initialization. Maximum program length is 200. Average results over 100 runs after 1000 generations.

Initial Length	CE mean	CE std.	Length abs.	Length eff.	Length %	Variations (%) constr.	Variations (%) neutral	Variations (%) noneff.
5	28.1	0.6	187	113	61	4.1	22	12
10	**25.7**	0.6	186	101	54	3.6	24	15
50	26.0	0.7	187	97	52	3.3	24	16
100	30.1	0.7	188	94	50	3.4	24	16
200	36.1	0.7	200	103	52	4.1	24	14

Table 6.20: *spiral*: Effects of initial program length on crossover (cross) using free initialization. Maximum program length is 200. Average results over 100 runs after 1000 generations.

by the initial program length.

When using effective mutations the noneffective code is not directly varied (effmut2) but may increase the effective step size indirectly by reactivations. This may be another reason why effective mutations perform worse here when applied to longer initial programs. The number of deactivations, instead, depends much less on the number of effective instructions. We will demonstrate in Section 8.7.2 that the effective step size increases less with the size of effective code than it is possible with the size of noneffective code. This effect becomes negative here such that smaller effective step sizes also reduce the variability of larger effective code.

In the experiments described above all initial programs share the same absolute length. One remaining question is whether variable-length programs in the initial population may produce significantly different results than constant-length programs. To answer that question, initial lengths are selected uniformly distributed from a range of 1 to 200 instructions in Tables 6.15 and 6.17. Thus, the average program length in the initial population, i.e., the total amount of genetic material, is about the same as in runs with (constant) initial length 100. In general, results show that a variable-length initialization changes the prediction error and the average complexity of programs only slightly compared to a constant-length initialization. This is mostly due to the fact that programs may still differ in their effective length even if their absolute length is constant. Only if the initialization is completely effective, variable (effective) lengths become more important and their effect on the performance is more significant (see Tables 6.16 and 6.18). Note, however, that there is hardly a relevant difference in performance here if the initial lengths are small.

6.7 Constant Program Length

In genetic programming usually programs of a variable-length representation are evolved. Typically, the population is initialized with smaller programs that grow in the course of the evolutionary algorithm. The traditional tree representation requires that programs change size and shape for creating successful solutions. Otherwise, if valid programs would be restricted to a constant number of nodes or a certain shape of a tree, variability and solution finding would be quite limited in general.

Constant	SSE		Length			Variations (%)		
Length	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
50	2.5	0.4	50	27	54	5.7	22	19
100	3.6	0.4	100	45	45	5.6	23	19
200	6.9	0.6	200	73	37	5.3	25	19

Table 6.21: *mexican hat*: Evolution of fixed-length programs using crossover (cross). Average results over 100 runs after 1000 generations.

Constant	CE		Length			Variations (%)		
Length	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
50	28.8	0.7	50	33	66	3.7	21	14
100	30.5	0.8	100	57	57	3.6	22	14
200	36.1	0.7	200	103	52	4.1	24	14

Table 6.22: *spiral*: Evolution of fixed-length programs using crossover (cross). Average results over 100 runs after 1000 generations.

The imperative representation used in linear GP contains inactive code that emerges almost independently from the composition of the provided sets of basic program elements (see Section 3.2). The only precondition for this special type of intron code is that the number of variable registers is larger than one. The existence of inactive code together with the fact that data flow between registers is organized as a graph allows an evolution of linear genetic programs without changing their absolute size. That is, programs may be initialized with a certain absolute length which stays constant during the whole run while only the effective length may change.

The evolution of fixed-length programs requires that the (absolute) program length is configured by the user instead of being subject to the evolutionary algorithm. This is a drawback because, first, the absolute length may have a significant influence on the prediction performance. Second, programs have a maximum size already from the beginning of a run. Thus, using a constant absolute program size is a combination of a maximum initialization and a restriction of program length. Both techniques have been investigated separately in the two previous sections. In this section we will verify for different variation operators whether a constant program length may be a feasible alternative to a growing program length.

First, let us compare Tables 6.21 and 6.22 with Tables 6.13 and 6.14 from Section 6.5, on the one hand, and Tables 6.19 and 6.20 from Section 6.6, on the other hand. Because the *mexican hat* problem profits from both a complexity control through a smaller maximum program size and a higher diversity by longer initial code, a better performance is obtained here when using a smaller constant length than it has been found in both partial experiments. This is in contrast to the *spiral* problem, which is better solved with larger program bounds and with less initial material (smaller initial step sizes) when using unrestricted linear crossover. As with *mexican hat*, a better fitness occurs here with constant program size 50 than with 200.

Constant Length	SSE *mean*	SSE *std.*	Length *abs.*	Length *eff.*	Length *%*	Variations (%) *constr.*	Variations (%) *neutral*	Variations (%) *noneff.*
50	1.3	0.1	49	25	51	9.2	8	0
100	1.0	0.1	98	44	45	8.9	9	0
200	3.5	0.4	196	79	40	8.6	11	0

Table 6.23: *mexican hat*: Evolution of fixed-length programs using effective mutations (effmut2, B0). Small variations in average program length because single instructions are inserted or deleted (not exchanged). Average results over 100 runs after 1000 generations.

Constant Length	CE *mean*	CE *std.*	Length *abs.*	Length *eff.*	Length *%*	Variations (%) *constr.*	Variations (%) *neutral*	Variations (%) *noneff.*
50	19.9	0.7	48	40	83	3.0	15	0
100	20.1	0.6	98	75	76	2.3	19	0
200	22.3	0.6	197	136	69	2.0	23	0

Table 6.24: *spiral*: Evolution of fixed-length programs using effective mutations (effmut2, B0). Small variations in average program length because single instructions are inserted or deleted (not exchanged). Average results over 100 runs after 1000 generations.

We have seen in Section 6.6 that the absolute and the effective program lengths strongly depend on the initial amount of genetic material when using effective mutations. The performance has been affected in such a way that rather small initial programs produce the best results. This has been accredited to the small step sizes of instruction mutations. Not surprisingly, evolving fixed-length programs by instruction mutations performs not better in general than using initial programs of the same length only (compare Tables 6.23 and 6.24 with Tables 6.15 and 6.17 above). The *spiral* problem is handled even worse with the additional parsimony effect. Slight improvements (if ever) have been found here only with some configurations of program length for the *mexican hat* problem.

In general, the smaller the step size is adjusted for a variation operator the more precise the approximation to a (sub)optimum may be (exploitation). The drawback is, however, that the escape from such suboptima (exploration) may be more difficult, especially if the solution size has already become relatively large. The lower variability of longer initial programs or fixed-length programs might require another exploitation-exploration trade-off in terms of the mutation step size. Hence, a larger absolute step size than one instruction may be supposed to better balance approximation, on the one hand, and variability, on the other hand. However, effective segment mutations (effonesegmut, see Chapter 5.7.5) have not produced relevant improvements here for different maximum segment lengths of 2–10 instructions (not documented). Already in Section 5.11.4 multiple (effective) instruction mutations have not proven to be more successful (with a standard initialization). Conclusively, also for maximum initialized programs an absolute mutation step size of one instruction might be close to the optimum.

6.8 Summary and Conclusion

Different control parameters were analysed in this chapter for their influence on linear GP. Some important results are summarized in the following.

(1) The performance of linear GP strongly depends on the number of calculation registers. Smaller register numbers may restrict the expressiveness of programs while larger numbers may increase the search space of programs unnecessarily. The more registers are provided the more registers may be effective and the lower is the effectiveness degree of instructions. For the functional structure this means wider graphs with less connections per node. A medium register number produced the best prediction results. More tree-like structures – resulting from higher register numbers – were usually not optimum.

(2) An initialization of all registers with input values achieved better results in general than setting the additional (calculation) registers to constant initial values.

(3) The question whether a smaller or a larger population size leads to more successful solutions could not be answered clearly (if the same number of evaluations is observed). Instruction mutations showed a significantly better performance in small populations for certain problems. Basically, this depends on the size of the optimum solution. In a smaller population programs grew larger, especially if the variation step size was small.

(4) Moreover, the relation of program size and fitness determines how much a problem solution profits from a higher complexity bound. When using (unrestricted) recombination linear programs grow noteably fast until the maximum length is reached. This was true for both the noneffective code and the effective code in programs, even if a larger upper bound led to a smaller *proportion* of effective code. We demonstrated by example that even large settings of the maximum program length may still produce better results.

(5) Finally, we compared possible initialization methods for linear genetic programs, including *maximum* and *fully effective*. In general, effective instruction mutations performed worse with a larger initial size of programs. Apparently, small absolute step sizes are less suitable to transform larger random structures. This was different for unrestricted segment variations which may perform better with more initial code.

(6) Linear GP allows the evolution of fixed-length programs, too, since programs may still vary in both their functional structure and their effective complexity. This may not be recommended in general, however, because it requires fixing the absolute solution size in advance and starting with initial programs of maximum length.

Chapter 7

A Comparison with Tree-Based Genetic Programming

Contents

A comparison between the linear representation and the traditional tree representation of genetic programs is performed in terms of both prediction performance and model size. The comparison is based on two collections of benchmark problems that have been composed of artificial test problems and real-world applications from bioinformatics, respectively. Both linear GP and tree-based GP use crossover for macro variations. Additionally, we apply the linear GP variant from Section 5.10.4 that works exclusively with (effective) instruction mutations. First of all, we will introduce tree-based GP into further detail.

7.1 Tree-Based Genetic Programming

The earliest and most commonly used approach to genetic programming is the evolution of tree structures represented by variable length expressions from a functional programming language, like S-expressions in LISP [51] This classic approach is referred to as *tree-based genetic programming* (TGP) for a better distinction from later approaches. The inner nodes of such program trees hold *functions* (instructions). The leafs are called *terminals* that mean input variables or constants.

In comparison with the imperative representation used in linear GP, pure functional programs, by definition, do not include assignments to memory variables. These have to be incorporated explicitly by means of special functions which realize read and write access to an external memory [51, 93]. Such "imperative" extensions are, however, not very commonly used because they do not always promise a higher functionality in a functional program. Nevertheless, assignments may be used if a tree program is supposed to return more than one output. Alternatively, multiple program outputs may be implemented in functional programs by using individuals that include multiple expressions (trees) which calculate one output each.

In any case, memory (a stack) is needed during the interpretation of program trees to save the intermediate results of each evaluated subtree (see also Section 3.3.3). While a program tree is evaluated the nodes are traversed in a predefined order (preorder or postorder). The value of a node is calculated by applying its function to all subtree results that have to be evaluated first. Then the value is returned to its father node. At the end of execution the tree root provides the final program output.

7.1.1 Genetic Tree Operators

Crossover is a genetic operator for recombining old solutions into new and potentially better solutions. Figure 7.1 illustrates representation and crossover in tree-based GP. In each parent individual the crossover operator selects a node (*crossover point*) randomly and swaps the two corresponding subtrees to create two offspring individuals. In general, the crossover points might be directed to function nodes for a higher probability than to terminal nodes. Koza proposes a 90 percent selection of inner nodes here.

The mutation operator exchanges single terminals or function identifiers. Usually each tree node is selected as a mutation point for the same probability. A *node mutation* replaces a random function by a legal alternative from the function set that requires the same number of parameters. In doing so, loss or creation of complete subtrees are avoided. This includes that functions may not be replaced by terminals (zero parameters) and vise versa. A certain amount of constants is maintained in tree programs by setting constant terminals for a user-defined proportion.

Alternatively, during *subtree mutation* a complete subtree is replaced by a random one. For the creation of the new subtree the same method may be applied as for the initialization

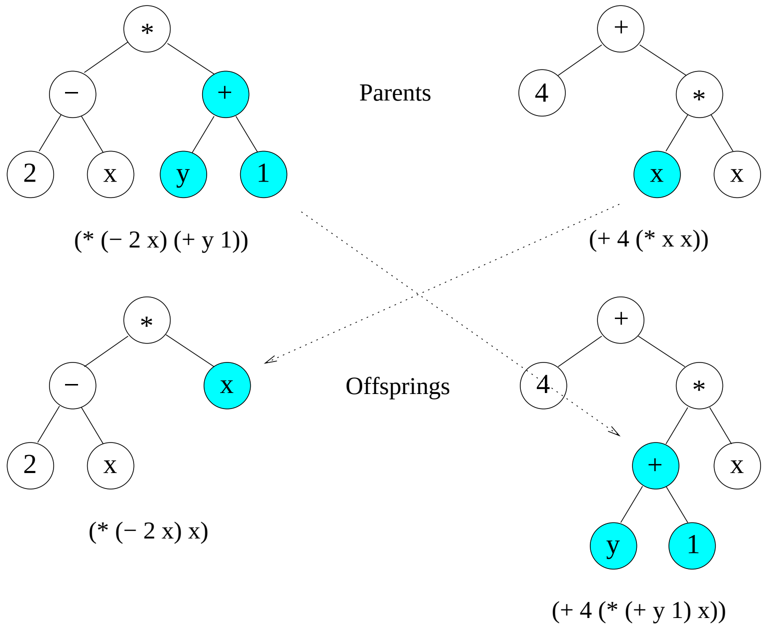

Figure 7.1: Crossover in tree-based GP. Subtrees in parents are selected and exchanged.

of programs (see next section). In contrast to crossover it has to be explicitly guaranteed that subtree mutations are bias-free. This is true if inserted subtrees are of the same size as deleted subtrees, on average.

In the standard TGP approach crossover is aborted, i.e., the effects of this genetic operator are reversed, if one of the offsprings violates a maximum complexity bound. Since the parent individuals are not modified in this case, they neither have to be reproduced nor their fitness has to be re-evaluated. The *maximum depth* of a tree denotes the maximum length of a path from the root to a leaf. If only the depth is limited as practiced by Koza [51] (who proposes a maximum depth of 17) programs may still become extremely large in their number of nodes, especially if a larger number of generations is observed. Moreover, the number of program nodes depends strongly on the average number of arguments that are required by the program functions. To better avoid a wasting of evaluation time and memory space a maximum limit may be placed on both the number of nodes and the depth of tree programs. For a fair comparison with linear GP it is necessary that the same maximum number of operations is observed in programs.

When using a smaller maximum number of nodes, individuals reach their maximum complexity more quickly. To assure that crossover remains executable, only equally large subtrees might be exchanged after (unrestricted) tree crossover has failed. This, however, would restrict the selection of variation points drastically. A better alternative might select a random successor node of the crossover point repeatedly (as new crossover point) until the corresponding subtree fits into the *other* parent. In order not to exchange smaller subtrees more frequently the unique path of predecessor nodes might be pursued from the crossover point for the same probability. That is, either the larger subtree is pruned or the smaller subtree is enlarged. This method leads to an exchange of equally large subtrees only if both parent individuals have maximum size.

We decided for a variant here that restricts the freedom of variation least by executing the crossover operation in any case. If an offspring tree becomes too large in terms of the number of nodes, the node at the crossover point is replaced by one of its direct successors (*after* crossover). The old node and all its other subtrees are deleted. This step is repeated recursively until the total tree size falls below the maximum bound. In contrast to selecting valid subtrees already in the parent individuals (*before* crossover) the positions of crossover points are freely selected here..

Recall from Section 5.7.1 that crossover in linear GP is always possible by exchanging equally-long instruction segments if otherwise the maximum program length would be exceeded. This is mostly due to a higher variability (weaker constraints) of the imperative representation which allows the existence of code that is not connected to the program output on the functional level (see Chapter 3).

In general, using genetic programming without any complexity bound is rather uncommon since unnecessarily large solutions are not desirable. First, those are less flexible during genetic manipulations. Within a certain number of generations reasonable progress may only be made up to a certain complexity of solutions. Otherwise too complex variations would be necessary to find successful solutions. Second, larger programs increase the processing time of GP and when being used in an application area. Third, interpretation of larger solutions is potentially more difficult. Finally, the principle of Occam's Razor says that shorter (equally fit) solutions are more general than longer ones. For these reasons low complexity is an important quality of genetic programs, besides a high prediction performance.

7.1.2 Initialization of Tree Programs

Genetic programs are created randomly from elements of the function set and the terminal set. However, even though these selections are random some methods are distinguished for tree-based GP that control the composition of genetic material in the initial population.

The *full method* generates only full trees, i.e., trees which have all terminal nodes on the same level. Another way to say this is that the tree path length from any terminal node to the root of the tree is the same.

The *grow method* chooses any node (function or terminal) for the root, then recursively calls itself to generate child trees for any nodes which need them. If the tree reaches the maximum depth, all further nodes are restricted to be terminals, so growth will cease. The shape and size of trees strongly depends on the probabilities for which a terminal node or a function node is selected. Usually these probabilities are supposed to be equal.

The *half-and-half method* merely chooses the full method 50 percent of the time and the grow method the other 50 percent. All of the generation methods can be specified with a "ramp" of initial depth values instead of using the same depth. For instance, if the ramp is 2–5, then 25 percent of the trees will be generated with depth 2, 25 percent will be generated with depth 3, and so on. Note that the latter two methods, when called to generate a tree of depth n, can produce a tree with actual depth less than n. *Ramped half-and-half* is typically the method of choice for initialization since it produces a wide variety of tree shapes and sizes.

7.2 Benchmark Problems

Basically, the benchmark problems that have been composed here for the comparison with tree-based GP comprises three problem classes. These are classification, regression, and Boolean functions.

In general, a GP benchmark may be regarded as a combination of problem (data set) and instruction set. The difficulty of a problem strongly depends on the composition of the function set in GP since this set may, in principle, hold any function – including the optimum solution of a problem if this is known (trivial case). At least for artificial benchmark problems where the optimum solution is already known in advance the absolutely best configuration is not always desired. Instead, the problem difficulty is scaled over the provided set of elementary functions. An optimization of the function set may be interesting only in terms of real applications or if we want to compare the performance of GP with other methods.

7.2.1 GP Benchmarks (GPProben)

The first composition of problems tested here is referred to as GPProben. Some problems became popular benchmarks in the GP community or in the machine learning community. Others have already been used in experiments of this document, but not necessarily with the same configuration. Table 7.1 summarizes all relevant problem characteristics and problem-specific configurations. These comprise the dimensions of the data sets, on the one hand, and the fitness function and the function set, on the other hand.

Problem	#Inputs	Input Range	Output Range	#Fitness Cases	Fitness Function	Function Set
11multiplexer	11	$\{0,1\}$	$\{0,1\}$	2048	SE	$\{\wedge, \vee, \neg, if\}$
even5parity	5	$\{0,1\}$	$\{0,1\}$	32	SE	$\{\wedge, \vee, \neg\}$
even8parity	8	$\{0,1\}$	$\{0,1\}$	256	SE	$\{\wedge, \vee, \neg, if\}$
two chains	3	$[-2,2]$	$\{0,1\}$	500	CE	$\{+,-,\times,/,sin,cos,if >\}$
spiral	2	$[-2\pi,2\pi]$	$\{0,1\}$	194	CE	$\{+,-,\times,/,sin,cos,if >\}$
double sine	1	$[0,2\pi]$	$[-1,1]$	100	SSE	$\{+,-,\times,/\}$
distance	6	$[0,1]$	$[0,1]$	300	SSE	$\{+,-,\times,/,\sqrt{x},x^2\}$
mexican hat	2	$[-4,4]$	$[-1,1]$	256	SSE	$\{+,-,\times,/,e^x,x^2\}$

Table 7.1: Complexity and configuration of GPProben problems. Maximum input and output ranges are rounded. The set of constants is $\{0,1\}$ for Boolean problems and $\{1,..,9\}$ otherwise.

Among the Boolean functions, the 11multiplexer function calculates 1 of 8 input bits as output value that is singled out by 3 address bits [51]. The evenNparity functions ($N = 5$ and $N = 8$ here) compute 1 if the number of set input bits is even, otherwise the output is 0. Note that the lower-dimensional parity problem is treated without Boolean branches here. The fitness function for Boolean problems is the sum of output errors (SE).

The two classification problems spiral and two chains are described in Section 5.8.1 and Section 10.4.1, respectively. For all classification problems in this chapter the classification error (CE) defines the program fitness. The classification method is always *interval classification*: A program output $gp(\vec{i_k})$ is considered as correct for an input vector $\vec{i_k}$ if the distance to a defined class identifier $o_k \in \{0,..,m\}$ is smaller than 0.5, i.e., $|gp(\vec{i_k}) - o_k| < 0.5$.

The one-dimensional regression problem double sine requires the sine function to be approximated by arithmetic functions only over an input range of two periods. For a description of the two-dimensional regression mexican hat and the six-dimensional distance problem see Section 5.8.1 again.

7.2.2 Bioinformatics Problems (BioProben)

The second benchmark set BioProben that is tested here contains real-world classification problems which mostly originate from the *UCI Repository of Machine Learning Databases* [15]. All problems have a biological background in common. Typically the problem data features a high input dimension. The original data sets have only been edited slightly.

Splice junctions are points on a DNA sequence at which "superfluous" DNA is removed during the process of protein creation in higher organisms. The splice junction data set is composed of sequences of 60 nucleotide positions extracted from primates DNA. The problem represented by this data set is to recognize the boundaries between exons (the parts of the DNA sequence retained after splicing) and introns (the parts of the DNA sequence that are spliced out). Actually, the problem consists of two subtasks: recognizing exon/intron boundaries (referred to as EI sites), and recognizing intron/exon boundaries (IE sites). In the biological community, IE borders are referred to as *acceptors* while EI borders are referred to as *donors*. About 50 percent of the data comprise non-splice examples that have been taken from sequences known not to include a splicing site at all. The nominal attribute values A, G, T, and C – representing the four nucleotide bases from which the DNA code is built – have been replaced by numeric values here (see Table 7.2). The very few unknown or uncertain characters are represented by 0. The problem comes with three data sets, one for each class. The first 50 percent of each set is used for training, the following 25 percent for validation, and the last 25 percent for testing (see below). A second data set splice junction 2 is derived by excluding all non-splice examples. This results in the simpler task to distinguish IE sites from EI sites only.

Problem	#Inputs	#Classes	Input Range	Output Range	#Fitness Cases
splice junction	60	3	$\{1, .., 4\}$	$\{0, 1, 2\}$	1594
splice junction 2	60	2	$\{0, .., 3\}$	$\{0, 1\}$	768
promoters	57	2	$\{0, .., 3\}$	$\{0, 1\}$	106
ecoli	7	8	$[0, 1]$	$\{0, .., 7\}$	336
helicases	25	2	$[0, 1]$	$\{0, 1\}$	78
soybean	35	19	$\{0, .., 6\}$	$\{1, .., 19\}$	307
wine	13	3	*continuous*	$\{1, .., 3\}$	178
dermatology	34	6	$\{0, .., 3\}$	$\{1, .., 6\}$	366

Table 7.2: Complexity of BioProben data sets. For all these classifications problems a common fitness function (CE) and function set $\{+, -, \times, /, x^y, if >, if \leq\}$ are used.

Another problem that deals with the classification of DNA sequences is promoters. A *promoter* initiates the process of gene expression, i.e., the biosynthesis of a protein. The task is to predict whether subsequences of E. Coli DNA belong to a region with biological promoter activity or not. Each subsequence holds 57 nucleotides.

The task defined by the data set, which is referred to as ecoli in the UCI repository, requires the cellular localization sites of proteins in E. Coli bacteria to be predicted from several measured values. In doing so, eight classes (localization sites) have to be discriminated.

Helicases is a problem of electron microscopy image classification concerning the classification of two different structures of hexametric helicases of DNA strangs [23].

A diagnosis of 19 different soybean diseases has to be learned from the soybean data. 13 percent of the data samples suffer from missing input values which all have been completed here by constant 7. The task described by the wine problem is to differentiate between three sorts of wine by their constituents resulting from chemical analysis.

The last problem tested here comes from a medical domain. The differential diagnosis of erythemato-squamous diseases is a real problem in dermatology. The dermatology data set constructed for this problem domain requires the distinction of six diseases that all share most clinical and histopathological features of erythema with only very little differences.

Table 7.2 summarizes all features of the BioProben data sets, including the input dimension, the number of output classes, and the number of (training) samples. Most input ranges are discrete and comprise possible states of attributes.

7.2.3 Generalization Data

The most important capability of a prediction model is the generalization from a given set of input-output examples to unknown (unlabeled) data inputs. The generalization ability strongly depends on the correlation of training data and generalization data, too. Especially, in complex or higher dimensional data spaces there is a higher probability that the correlation between two randomly selected sets of data points is poor. Moreover, the generalization performance is influenced by the size of the training set and how regularly the training data is distributed over the problem data space.

This is especially true when dealing with data that is derived from a real application domain. The use of artificial test problems may give a better understanding for what types of problems a method is suitable and for what types it is not. Because the problem structure is usually known in advance, artifical benchmarks give a better idea of the problem difficulty, too. that may even be scalable. In general, a problem description defines the domain of the input data and the output data, including dimension and attribute ranges. Generalization data originates from the same data domain as the training data. The larger the data domain is, however, or the less it may be restricted by the user's knowledge, the more likely the data in both sets may cover different subspaces.

The identification of generalization data is obvious for the tested regression problems. With such continuous problems generalization means interpolation. For the mexican hat problem random points are selected that lie between the regular grid of training data points (see Figure 5.2). For the distance problem generalization data is created like the training set by calculating the Euklidean distance for random pairs of 3-dimensional points from defined input ranges.

Since we do not want to separate two clouds of fixed data points only in case of the two chains problem (see Section 10.4), the data space is supposed to include all points that lie within a certain distance from two virtual circles in three-dimensional space.

Concerning real-world problems the data domain often comprises much more examples than these may be represented sufficiently by the available data. Moreover, the available amount of data is very often limited. But even if this is not true, data samples may hardly be selected uniformly distributed because the structure of the data space is usually unknown. As a result, correlation of training data and generalization data may be low.

In some data sets of the BioProben collection, e.g., ecoli, the distribution of data examples over the classes is quite non-uniform such that some classes are represented by only a few examples. With other data sets the number of examples is relatively small compared to the number of inputs, e.g., promoters. In both cases it is rather difficult to split a data set for training, validation and testing. Results may strongly depend on random rather than being general. One possibility to get more reliable generalization results is to apply *n-fold cross validation* in such cases, a method that divides the data into n disjoint subsets and repeats a learning process n times while each time another subset is excluded from training and used for validation. This has not been practiced here, however. Instead, we restrict ourselves to the splice junction problem.

According to the evolutionary algorithm in Section 2.3 the generalization ability of the best-so-far individual is checked during training by calculating its error on a validation set. At the end of a run the individual with minimum validation error is applied again on a test set. Except for the spice junction problem, both the validation set and the test set contain about as many examples each as the training set (see Table 7.1).

7.3 Experimental Setup

A comparison between completely different methods, as performed for neural networks and genetic programming in Chapter 4, may be based on the prediction performance only. In this case, simply the best or nearly the best configuration may be selected for each approach. If the tested approaches are more related, however, similar parameters should be configured similarly to guarantee a fair comparison. This is the more important the less two approaches differ. Otherwise, the different feature(s) may hardly be made responsible for a potential difference in performance. For the same reason comparing results from literature may be crucial. Comparability of results may be guaranteed best within the same system environment.

7.3.1 A Multi-Representation System

Most experiments in this chapter have been performed with a multi-representation GP system [18] that comprises different representation forms for genetic programs, including trees and linear structures in particular. Such a platform allows the user to test different representation types with only a minimum implementation overhead, i.e., without changing the adaptation of the system to a certain problem. A most fair comparison of GP representations is achieved by using the same system environment as far as possible. Among other things, that includes a common evolutionary algorithm, a common selection method, and a general definition of instruction set and terminal set. In this way, the probability is reduced that slightly differing implementation details or parameter configurations may influence the results.

7.3.2 Complexity of Programs

The following comparison between the tree representation and the linear representation of genetic programs has been tried to be as fair as possible. First of all, the comparison is fair in terms of the maximum complexity of programs. If we assume that all program parts are executed this is true for the evaluation time as well. In particular, the same maximum number of instructions (200 here) is allowed in both kinds of programs. For program trees this is the number of inner (non-terminal) nodes while in a linear program the number of lines is counted. The lower bound of absolute program size corresponds to one instruction (node).

Alternatively, it may be taken into account that not all instructions of the linear representation – in contrast to a tree representation – are structurally effective (after Definition 3.4). Remember that such noneffective instructions may always be removed completely from a linear program before it is executed and, therefore, do not cause computation costs (see Section 3.2.1). Thus, the actual solution is represented by the effective program only.

From that point of view, it may be a legal alternative to restrict the *effective* length of a linear program instead of its absolute length. This may be realized, for instance, by repeating a crossover operation until a maximum number of effective instructions is met. In so doing, a maximum of n inner tree nodes is regarded as being equivalent to n effective instructions. Such a comparison would still be fair in terms of the same maximum number of *executed* operators. Only the total number of operators, i.e., the absolute complexity, may be larger for linear programs than for tree programs.

The maximum (absolute) length may not be left completely unrestricted, however. First, a higher amount of noneffective code usually implies a larger effective code, too. Second, the absolute (and the effective) crossover step size are increased because longer segments are exchanged. Finally, it has to be noted, that longer (effective) programs do not always provide better solutions for a problem (see Section 6.5). Not only for these negative effects the absolute program length is better limited sufficiently in linear GP.

The reader may remember from Section 5.9.1 that there is another argument for restricting the absolute program length and leaving the effective length to be influenced only indirectly by this. The structurally noneffective code takes away a lot of pressure from the effective code to grow and to develop semantic introns as a protection against larger crossover steps. In other words, the presence of noneffective code puts an *implicit parsimony pressure*[1] on the effective code size which does not have to become much larger in this way than necessary for a solution's fitness. Therefore, the structural noneffective code, that may be detected and removed completely, is another reason why effective LGP solutions may be more compact in size than TGP solutions.

Furthermore, linear genetic programs may manage-with a smaller number of operations because their functional structure is a directed acyclic graph (DAG), i.e., is not restricted to a tree structure. Among other things, the higher freedom of connections between the program functions allows the result of subsolutions (subgraphs) to be reused multiple times. We may not automatically conclude, however, that the expressiveness of a DAG program is higher than the expressiveness of a tree program. First, the same functionality (instruction set) is provided for both types of representation. Second, a DAG can be transformed into a tree and each tree is a special DAG. Nevertheless, if the same maximum number of instructions is allowed imperative programs may express more complex solutions than tree programs.

If a comparison of program complexity would be based on *all* nodes – including terminals – instead of counting the (effective) function nodes only, the differences in size between tree programs and DAG programs would be even more significant.

A binary tree structure with n inner nodes may have $n + 1$ additional terminal nodes at maximum. In the imperative representation *terminals* may be defined as all constant operands and all read-outs of registers before their original content is overwritten for the first time in a program. The number of *constant terminals* is bounded by the number of instructions (n). Recall that we allow at most one constant operand per instruction. The number of *variable terminals* in an (effective) linear program is usually significantly smaller. Especially if the register set is rather small overwriting of information happens easily in programs. In the DAG representation of a linear program the number of terminals is bounded by the sizes of register set and constant set (see Section 3.3), i.e., each register or constant is represented at most once.

[1] For an *explicit* parsimony pressure see Section 9.9.3.

7.3.3 Parameter Settings

Table 7.3 lists the parameter settings for both GP approaches. The parameter settings are supposed to be general and have not been adapted to a specific problem. Parameters that are necessarily problem-dependent like the fitness function and the function set have been introduced together with the benchmark problems in the previous section

More general conclusions may be drawn about the performance of the two GP variants, if especially parameters that exist for one variant only are not explicitly optimized for each problem. For linear GP we allow 10 additional registers besides the required minimum number of (writable) registers that hold the inputs. All registers are regularly initialized with input data such that each input value is assigned to about the same number of registers. We have seen in Chapter 6 that both may have a significant influence on the performance of linear GP, especially for problems with a lower input dimension. Only if the number of inputs is already much larger than 10, as in case of most BioProben tasks, no additional registers are provided. In this case the total number of registers may be sufficiently high while additional registers unnecessarily increase the search space only.

The average size of an initial program comprises about 20 operations in all experiments. In particular, as many instructions are used in initial linear programs as there are (inner) nodes in initial tree programs, on average. In linear GP this is realized simply by choosing the initial program lengths uniformly distributed from an appropriate range. In tree-based GP we apply the ramped-half-and-half method (see Section 7.1.2). which may be controlled by a maximum and a minimum depth of initial trees. This results in a more probabilistic configuration of initial program size in terms of the number of operator nodes. Note that the maximum number of nodes in a tree of a certain depth depends on the arity of instruction nodes, too. Therefore, the maximum number of nodes in initial programs may be restricted additionally.

LGP	
Parameter	Setting
Number of generations	500 (1000)
Population size	500
Maximum program length	200
Initial program length	10–30
Initialization method	random
Number of registers	#inputs + 10
Macro variation	90%
Micro mutation	10% (100%)
Selection method	tournament
Tournament size	2
Instructions with constant	50%

TGP	
Parameter	Setting
Number of generations	500 (1000)
Population size	500
Maximum operator nodes	200
Maximum tree depth	17
Initial tree depth	4–7
Initialization method	ramped
Crossover	90%
Node mutation	10% (100%)
Selection method	tournament
Tournament size	2
Constant Terminals	25%

Table 7.3: General parameter settings for linear GP (left) and tree-based GP (right).

A balanced ratio of population size and generations has been chosen to guarantee both a sufficient number of evaluations per individual and a sufficient diversity. In general, the population size should not be too large in relation to the total number of evaluations trained. Otherwise results depend less on the evolutionary progress made by the genetic operators but more on random effects. A too small population, instead, would make the performance depend more strongly on the composition of the initial genetic material, especially when using crossover (see also Section 6.4).

Besides similar complexity bounds, the same steady state EA is used, as described in Section 2.3, including the same selection method (tournament selection). Genetic operators are highly specific for a each representation type, of course. Exactly one genetic operation is executed per individual. Both linear crossover and tree crossover are unrestricted in terms of the maximum size of exchanged subprograms.

As noted before, linear crossover, while operating on the imperative code, may affect multiple crossover points on the functional level. In contrast to that, tree crossover always affects one crossover point. For that reason crossover may be considered to be more destructive in linear GP. On the other hand, small pieces of code may be exchanged at all parts of the linear representation. Moreover, (structural) introns may easily be created at each position in a linear program to reduce the effective step size. For tree representations both is more difficult to achieve especially in upper regions (see Section 7.5).

All variations including both crossover and mutations are induced effectively for linear GP. That is, each genetic operation alters at least one effective instructions. Remember that operations on program trees are fully effective in this meaning because structural noneffective code is not defined (see Section 3.2). For linear crossover it is sufficient to guarantee the effectiveness for the deleted segments (effdel, see Section 5.7.4). Then noneffective crossover variations may only result from (effectively) identical exchanges of code which are not avoided explicitly here because these are usually not very likely. Additionally, we compare a pure mutation-based variant of LGP that applies effective mutations (effmut2, B1, see Section 5.10.4) as a macro operator with a minimum segment length (one instruction).

There are only two differences between the parameter configurations used for GPProben and BioProben. First, a twice as large population size of 1000 individuals is used in the latter collection of benchmark problems. This implies twice as many evaluations of individuals for the same number of generations. Second, since the average input dimension is significantly higher for most BioProben tasks, micro (node) mutations are applied for 100 percent, either in combination with crossover or not. Both a larger population size and a high mutation rate guarantee a higher instruction diversity in the population. Note that especially a high input (register) number leads to more possible combinations of instructions.

As a third countermeasure, the initial average size of programs should be large enough to guarantee a sufficient number of effective instructions. Too few instructions may result in many identical programs if a problem requires many (input) registers. A similar effect might be achieved by a fully effective initialization here (see Section 6.6).

7.4 Results and Comparison

7.4.1 Prediction Quality and Complexity

Tables 7.4 and 7.5 show the performance of the best solution of a run that has been found with tree-based GP and with linear GP, respectively, for the GPProben collection of test problems. In TGP program size is given by the number of operator nodes and by the tree depth. In LGP the absolute and effective program length are differenced. Each complexity measure is averaged over all programs of a run. Because the execution of programs during the fitness calculations is by far the most time-consuming step, the average (effective) complexity is directly related to the computational overhead of a GP variant. All results are again averaged over 100 independent runs.

When comparing the prediction errors of both GP approaches, most test problems are better solved by linear GP (except for distance). In general, the difference is most clear for the discrete problems here, including Boolean problems and classifications. In particular, much higher hit rates have been found with 11multiplexer, even8parity and two chains. Among the continuous (regression) problems the difficult mexican hat problem is treated significantly better by means of an imperative representation.

In all test cases the size of tree programs occurs much larger in Table 7.4 than the effective program length in Table 7.5. Because both measurements count the number of executed instructions, they may be directly compared here. The average absolute length of linear programs is similar for all problems and comes typically close to the maximum limit of 200 instructions. As argued in Section 7.3, linear (effective) solutions may be more compact due to both the existence of structural noneffective code and the underlying graph structure that allows a multiple reuse of code.

Compared to unrestricted linear crossover, Table 7.6 documents a much higher prediction quality in linear GP for all eight test problems when using mutations with a minimum segment length. Especially for most discrete problems, not only the average prediction error is significantly smaller, but the optimum solution has been found in much more runs, too. Since variations are always effective here, the worse performance of linear crossover compared to instruction mutations may be accredited mostly to the difference in step size.

A parsimony effect of both the maximum program length and the noneffective code are responsible for the very similar *effective* size of solutions that has been found with crossover in Table 7.5 and with instruction mutations in Table 7.6. This might be an evidence, too, that the proportion of semantic introns in effective programs is rather small. At least, it shows that a difference in (effective) program size may hardly be responsible for the large difference in prediction quality here.

The prediction quality and complexity of solutions, that have been found for the BioProben collection of (classification) problems, is printed in Tables 7.7 to 7.9. As already observed with discrete problems from GPProben, for all BioProben problems the average performance is higher with a linear representation. Concerning the quality of best solutions this is only true for the splice junction problem. In all other problem cases the best errors are similar.

The higher best and average prediction performance that has been found with effective mutations demonstrates again that this operator outperforms linear crossover clearly. This is true even if the improvements are relatively smaller for the real-world problems here, on average, than for the GPProben benchmarks.

For some problems the average effective length grows significantly larger when using instruction mutations than this has been found with linear crossover (compare Tables 7.8 and 7.9). One explanation is that a certain amount of noneffective code will always emerge with crossover. Depending on the maximum bound this restricts the growth of effective code (as noted above). Another explanation is the relatively high input dimension of BioProben problems which requires many registers. Since the applied mutation operator creates each new instruction effectively, the proportion of effective code is much more independent from the number of registers (see Section 6.1).

Problem	Error			#Hits	Size	Depth
	best	***mean***	*std.*			
11multiplexer	0.0	186.0	12.1	10	138	15
even5parity	2.0	8.3	0.2	0	143	15
even8parity	0.0	68.6	2.1	1	179	11
two chains	0.0	13.4	1.1	5	146	15
spiral	17.0	36.0	0.9	0	152	15
double sine	0.2	8.7	0.8	0	147	15
distance	0.0	6.8	0.5	0	68	13
mexican hat	0.5	11.6	1.1	0	81	14

Table 7.4: GPProben: Prediction quality and program size using crossover-based TGP. Average results over 100 runs after 500 generations. Average program size given in operator nodes.

Problem	Error			#Hits	Length		
	best	***mean***	*std.*		*abs.*	***eff.***	%
11multiplexer	0.0	92.0	9.1	31	189	88	46
even5parity	1.0	8.4	0.3	0	173	46	26
even8parity	0.0	25.9	2.2	22	167	88	52
two chains	0.0	4.7	0.5	24	186	79	42
spiral	7.0	24.6	0.5	0	187	87	46
double sine	0.6	7.7	0.7	0	181	48	27
distance	0.6	8.7	0.3	0	185	31	17
mexican hat	0.05	3.2	0.3	0	189	37	19

Table 7.5: GPProben: Prediction quality and program size using crossover-based LGP. Average results over 100 runs after 500 generations.

Problem	Error			#Hits	Length		
	best	***mean***	*std.*		*abs.*	***eff.***	%
11multiplexer	0.0	2.3	1.1	94	101	83	82
even5parity	0.0	1.3	0.1	38	77	43	55
even8parity	0.0	1.6	0.3	68	101	85	84
two chains	0.0	0.8	0.1	50	96	77	80
spiral	0.0	10.4	0.4	1	93	80	86
double sine	0.04	2.9	0.3	0	76	45	59
distance	0.0	2.9	0.2	1	74	36	48
mexican hat	0.01	1.0	0.1	0	79	39	49

Table 7.6: GPProben: Prediction quality and program size using mutation-based LGP. Average results over 100 runs after 500 generations.

Problem	Error			#Hits	Size	Depth
	best	***mean***	*std.*			
splice junction	211.0	386.0	8.2	0	138	15
splice junction 2	14.0	36.1	2.2	0	137	15
promoters	0.0	5.8	0.6	2	142	15
ecoli	37.0	73.2	2.2	0	151	15
helicases	0.0	2.1	0.1	6	148	14
soybean	79.0	153.5	6.3	0	134	14
wine	0.0	17.4	1.5	2	147	14
dermatology	4.0	57.4	4.8	0	134	14

Table 7.7: BioProben: Prediction quality and program size using crossover-based TGP. Average results over 50 runs after 500 generations. Average program size given in operator nodes.

Problem	Error			#Hits	Length		
	best	***mean***	*std.*		*abs.*	***eff.***	%
splice junction	78.0	189.1	10.6	0	160	58	36
splice junction 2	6.0	18.4	1.1	0	163	66	40
promoters	0.0	1.7	0.2	8	181	54	30
ecoli	36.0	54.0	1.4	0	180	77	43
helicases	0.0	1.4	0.1	12	184	79	43
soybean	67.0	95.3	2.2	0	186	70	38
wine	0.0	2.5	0.2	3	138	87	63
dermatology	4.0	14.3	1.3	0	186	69	37

Table 7.8: BioProben: Prediction quality and program size using crossover-based LGP. Average results over 50 runs after 500 generations.

Problem	Error			#Hits	Length		
	best	***mean***	*std.*		*abs.*	***eff.***	%
splice junction	52.0	97.4	5.2	0	140	110	78
splice junction 2	5.0	11.9	0.7	0	127	104	82
promoters	0.0	0.3	0.1	30	111	89	80
ecoli	22.0	32.2	0.8	0	98	86	88
helicases	0.0	0.7	0.1	36	105	87	83
soybean	30.0	55.6	2.4	0	111	94	84
wine	0.0	1.2	0.1	9	118	103	87
dermatology	2.0	4.3	0.3	0	112	92	82

Table 7.9: BioProben: Prediction quality and program size using mutation-based LGP. Average results over 50 runs after 500 generations.

7.4.2 Generalization Ability

The generalization results for the regression problems in Tables 7.10 to 7.12 demonstrate that both the validation error and the test error come very close to the training error (in Tables 7.4 to 7.6). That is, a variation operator that improves the training performance improves the generalization results for almost the same amount here. In such a case we may assume that the correlation between training data and generalization data is high.

The generalization errors of the tested classification problems may be significantly different from the training error, especially when using effective mutations.[2] This may be accredit to the use of branches here. In general, branches improve the training performance such that they support a specialization to certain training examples. Without using branches the three prediction errors would become more similar (not documented). Nevertheless, both validation error and test error are smaller with branches. At least, if branches are essential for finding the optimum solution or guarantee a significantly higher fitness they may not lead to a worse generalization quality. Another reason may be that training data and generalization data are less correlated for a problem.

For the same reason the generalization errors are more similar than the training errors when comparing different GP representations, on the one hand, or different genetic operators, on the other hand. Obviously, a genetic operator or a representation that performs better than others on the training set may not necessarily do the same on unknown data if this originates from a too different region of the data space.

7.5 Discussion

Instruction mutations vary the length of the imperative code in minimum steps. On the functional level only one operator node is inserted in or deleted from the corresponding program graph, together with its incoming and outgoing edges. First, because the degree of freedom is higher in a directed acyclic graph than in a tree, by definition, the imperative representation allows insertions or deletions of code to be permanently small at each position.

Second, code parts may become structurally noneffective in linear programs. That means they may be disconnect only temporarily from the effective graph component (see Section 3.3). Instruction mutations as applied in this section do not avoid such disconnections (deactivations) of code explicitly (see also Section 5.10.5). On the one hand, the coexistence of inactive (disconnected) code in programs avoids an irrecoverable loss of code and allows its reactivation (reconnection). On the other hand, the graph structure allows multiple connections of nodes which reduces the probability for disconnections. Additionally, disconnections decrease implicitly in the course of a run as a result of an increasing connection degree of instruction nodes, as will be demonstrated in Section 8.7.2.

Both is different in tree-based GP. Due to the higher constraints of the tree structure deletions or insertions of subtrees are not possible as separate operations. A tree structure requires a removed subtree to be directly replaced at the same position. In linear GP the register identifiers (pointers) are encoded in the instructions. If those are disconnected from a subprogram by deactivation, they are either automatically reconnected to other instructions or represent a terminal.

In general, (macro) mutations that change the size and the shape of trees are less likely small on higher tree levels. At least deletions of larger subtrees may not be avoided without restricting the freedom of variation significantly. Since in a tree each node is connected

[2] Half as many data examples used during validation and testing than during training with splice junction.

Problem	Validation Error			#Hits	Test Error			#Hits
	best	***mean***	*std.*		*best*	***mean***	*std.*	
two chains	0.0	10.9	0.6	1	1.0	11.9	0.6	0
splice junction	130.0	208.3	3.7	0	144.0	212.2	3.4	0
distance	0.0	6.9	0.6	0	0.0	7.3	0.5	0
mexican hat	0.4	15.9	1.5	0	0.4	16.2	1.4	0

Table 7.10: Generalization ability using crossover-based TGP.

Problem	Validation Error			#Hits	Test Error			#Hits
	best	***mean***	*std.*		*best*	***mean***	*std.*	
two chains	0.0	7.9	0.5	2	2.0	8.4	0.4	0
splice junction	69.0	120.3	5.6	0	55.0	123.8	5.9	0
distance	1.4	10.3	0.3	0	1.2	9.6	0.3	0
mexican hat	0.03	3.3	0.4	0	0.03	3.6	0.5	0

Table 7.11: Generalization ability using crossover-based LGP.

Problem	Validation Error			#Hits	Test Error			#Hits
	best	***mean***	*std.*		*best*	***mean***	*std.*	
two chains	0.0	4.6	0.3	6	2.0	5.1	0.3	0
splice junction	59.0	88.7	3.1	0	57.0	89.7	3.2	0
distance	0.0	3.5	0.3	1	0.0	4.0	0.3	1
mexican hat	0.006	1.1	0.1	0	0.006	1.3	0.1	0

Table 7.12: Generalization ability using mutation-based LGP.

only by one edge on a unique path to the root and since the tree representation does not allow unconnected components, a disconnection of code always means its loss.

Nevertheless, the probability for such larger mutation steps may be reduced as far as possible in TGP. Therefore, three elementary tree operations may be distinguished – including insertion, deletion and substitution of single nodes (as proposed in [70]).

If an operator node is inserted in a tree it replaces a random node that becomes a successor of the new node (if required). All remaining successors of the newly inserted node become terminals. Since usually most instructions require more than one operand, almost each insertion will create a new terminal node, in this way. Accordingly, if a random inner node is selected for deletion it is replaced by one of its successors (or the largest subtree). All other successors of the deleted node get lost (including the corresponding subtrees). Finally, a node may be substituted by another node of the same arity only. This implies that terminal nodes are exchanged by other terminals only. In so doing, the tree structure is not changed by substitutions. Otherwise, if this is not practiced, supernumerary subtrees may be completed by a terminal or deleted, respectively.

In [70] such *minimum structural mutations* are applied in combination with search techniques like simulated annealing and hill climbing in GP both operating with a single search point (individual). In [71] the authors combine these search techniques with a standard population-based search by crossover. Unfortunately, the performance of these mutations is not compared with standard crossover and the same search method.

7.6 Conclusion

After an introduction to tree-based GP, we compared this more traditional approach with linear GP by using two collections of benchmark problems. The comparison was supposed to be fair particularly with regard to the (maximum) complexity of genetic programs.

(1) With unrestricted crossover LGP performed better than TGP and produced more compact solutions in terms of the number of executed instructions. Especially for (real-world) classification problems the difference in performance between a tree representation and a linear representation was most significant.

(2) Even better prediction results were obtained for linear GP by means of effective instruction mutations. This was especially true for the applied GP benchmarks. Here results showed a much smaller difference in performance between the two representation forms than between the two linear genetic operators applying maximum (unrestricted) or minimum step sizes. This recommends a general use of minimum mutation steps in linear genetic programming and confirms our results from Chapter 5 for a wider range of applications.

(3) We also argued why, first, LGP allows smaller solutions and, second, a minimization of grow and shrink operations may only be incomplete in TGP. Both may be reduced to the two fundamental differences of the representations that have been outlined already in Chapter 1. In the first case, this means that (effective) linear programs may be more compact in size because of a multiple usage of register contents and an implicit parsimony pressure by the structurally noneffective code. In the second case, the higher constraints of the tree structure and the lack of non-contiguous components avoid that structural step sizes may be permanently minimum.

Chapter 8

Explicit Control of Diversity and Variation Step Size

Contents

We will now investigate structural and semantic distance metrics for linear genetic programs. Causal connections between changes of the genotype and the phenotype form a necessary condition for analyzing structural differences between genetic programs and for the two major objectives of this chapter: (1) Distance information between individuals is used to control structural diversity of population individuals actively by a two-level tournament selection. (2) Variation distance is controlled probabilistically on the effective code for different linear genetic operators.

8.1 Introduction

In contrast to other evolutionary search algorithms, like evolution strategies (ES), genetic programming (GP) may fulfill the principle of *strong causality*, i.e., small variations in genotype space imply small variations in phenotype space [75], less strongly [78]. Obviously, changing just a small program component may lead to almost arbitrary changes in program behavior. However, it seems to be intuitive that the more instructions are modified, the higher is the probability of a large fitness change.

As discussed in Section 5.4, a *fitness landscape* on the search space of programs is defined by a structural distance metric between programs and a fitness function that reflects the quality of program semantics. The application of a genetic operator corresponds to performing one *step* on the landscape. In general, the variation step size should be related to a distance metric that constitutes a fitness landscape that is smooth at least in local regions.

The *edit distance*, sometimes referred to as *Levenshtein distance* [35], between varying length character strings has been proposed as a metric for representations in genetic programming [46, 72]. Such a metric not only permits an analysis of genotype diversity within the population but offers a possibility to control the step size of variation operators more precisely. In [41] correlation between edit distance and fitness change of tree programs has been demonstrated for different test problems. This chapter introduces efficient structural distance metrics that operate selectively on substructures of the linear program representation. Correlation between structural and semantic distance as well as distribution of distances are documented for different types of variation.

One major objective of this chapter is to control structural diversity, i.e., the average program distance, in LGP populations explicitly. Therefore, we introduce a two-level tournament that selects for fitness on the first level and for diversity on the second level. We will see that this is less motivated by a better *preservation* of diversity during run but by a control of a diversity *level* that is depending on the configuration of the selection method. We will also see that prediction improves significantly if the diversity level of a population is increased.

The simplest form of diversity control might be to seed randomly created individuals regularly into the population during runtime. In [46] a more explicit maintenance of diversity is proposed by creating and seeding individuals that fill "gaps" of under-represented areas in genotype space. However, experimental evidence is not given for this rather complicated and computationally expensive approach. Until now, explicit diversity control is a rarely investigated technique in genetic programming. Recently, de Jong *et al.* [44] could improve parsimony pressure through Pareto-selection of fitness and tree size by adding a (third) diversity objective. A more implicit control of genetic diversity, by comparison, offer semi-isolated subpopulation, called *demes*, that are widely used in the area of evolutionary computation (see also Section 4.3.2). Only a certain percentage of individuals is allowed here to migrate from a deme into another deme during each generation.

The second major objective of this chapter refers to the structural distance between a parent program and its offspring, i.e., the variation step size. While the effect on the absolute program structure, i.e. the absolute variation distance (Definition 5.3), may be controlled implicitly by the genetic operator, as demonstrated in Chapter 5, the amount of change induced on the effective code, i.e., the effective variation distance (Definition 5.4), may differ significantly from the absolute change. By monitoring the effective variation distance explicitly, structural step sizes are controlled more precisely in relation to their effect on program semantics. We will demonstrate that even strong restrictions of the maximum allowed effective mutation distance do not necessarily imply relevant restrictions of the freedom of variation, too.

We apply two different variants of linear GP in this chapter for macro variations. While the first approach applies recombination by linear crossover the other approach is based on (effective) instruction mutations (see Chapter 5). In the first case the absolute variation distance is unlimited while in the latter case it is restricted to a minimum.

8.2 Structural Program Distance

8.2.1 Edit Distance

The *string edit distance* [35] calculates the distance between two arbitrarily long character strings by counting the number of basic operations – including insertion and substitution of single elements – that are necessary to transform one string into another. Usually each operation is assigned the same costs (1) independently from the affected type of element. The standard algorithm for calculating the string edit distance needs time $O(n^2)$ with n denotes the maximum number of components that are compared between two individual programs. Recently, some more efficient algorithms have been presented [62].

We apply the edit distance metric to determine the structural distance between the effective part of programs since a difference in effective code may be more directly related to a difference in program behavior (semantic distance). In general, the correlation between semantic and structural distance is the more lower the higher the proportion of noneffective code is that occurs with a certain variation operator or parameter configuration. It is important to realize that the effective distance is not part of the absolute distance. Actually, two programs may have a small absolute distance while their effective distance is comparatively large (see Section 8.5). On the other hand, two equally effective programs might differ significantly in their noneffective code.

For an efficient distance calculation we concentrate on representative substructures of linear programs and regard only the sequence of operators (from the effective instructions). The sequence corresponding to Example 8.1 is $(-, +, /, +, *, -, -, /)$ when starting with the last effective instruction. The distance of effective operator symbols has been found sufficiently precise to differentiate between program structures provided that the used operator set is not too small. This is also due to the fact that in most cases the modification of an effective instruction changes the effectiveness status of at least one instruction. Note that in contrast to the effective distance the absolute operator sequence would not be altered by the exchange of single registers.

Because identical exchanges of program components are avoided updating a constant by another constant is the only type of variation that is not registered at all. In general, a registration of absolutely every structural difference should not be necessary if we take into account that the correlation between semantic and structural distance is probabilistic (see Section 8.7.1).

```
void gp(r)
  double r[5];
{
   ...
// r[4] = r[2] * r[4];
   r[4] = r[2] / r[0];
// r[0] = r[3] - 1;
// r[1] = r[2] * r[4];
// r[1] = r[0] + r[1];
// r[0] = r[3] - 5;
// r[2] = pow(r[1], r[0]);
   r[2] = r[3] - r[4];
   r[4] = r[2] - 1;
   r[0] = r[4] * r[3];
// r[4] = pow(r[0], 2);
// r[1] = r[0] / r[3];
   r[3] = r[2] + r[3];
   r[4] = r[2] / 7;
// r[2] = r[2] * r[4];
   r[0] = r[0] + r[4];
   r[0] = r[0] - r[3];
}
```

Example 8.1: Linear genetic program. Noneffective instructions are commented. Register `r[0]` holds the final program output.

Beyond that, less different genotypes are distinguished by this *selective* distance metric that represent the same phenotype (fitness). By including the program registers into distance calculation the distance measure might become even more ambiguous. Actually, most registers are used temporarily only during calculation and may be replaced partly by others without altering the behavior of a program. In fact, only the last assignment to an output register in (effective) program and all readings of an input register before its contents is overwritten for the first time are invariable. Additionally, the distance between operator sequences is not unique since the order of instructions may be changed without changing the program behavior, as indicated in Section 3.3.3. Nevertheless, a linear program may be sufficiently represented by its operator sequence. This is especially true since the functional dependencies between the instruction nodes usually form a rather narrow ("linear") graph structure (see Section 3.3). The more narrow the graph structure is the more the position of an operator corresponds to its position in the sequence (see Section 3.4).

Another important motivation for restricting the number of components in the compared programs is that time of distance calculation is reduced significantly. By regarding only the sequences of effective operators calculation time of edit distance directly depends on the (average) number n of effective instructions only. Depending on the percentage of noneffective code there are k times more elements to compare if one regards the full sequence of operators in programs. Extending the distance metric to registers and constants of instructions, again, results in a factor of 4 maximum. In conclusion, computational cost of the edit distance would increase by a total factor of $(4k)^2$ up to $O(16k^2 \cdot n^2)$.

Effective mutations, as introduced in Chapter 5, guarantee that the effective code will change. Such operations work closely with our effective distance metric here such that not more than one instruction is inserted, deleted or changed (maximum absolute distance 1). Recall that macro mutations operate on full instruction level, while micro mutations vary smaller components within instructions, i.e., operate below instruction level. In order to guarantee a sufficient variation and growth of programs, however, the higher number of variations is performed on macro level (see Section 8.6). Since, in this way, the absolute step size is not further reducible from operator side, measuring the distance between *full* (effective) programs, i.e., on micro level, does not necessarily promise a higher precision. This is another reason why operator sequences represent a sufficient basis for distance calculation between linear genetic programs.

8.2.2 Alternative Distance Metrics

In all the following experiments we have applied the edit distance metric as described above. However, even if a reduction of identifying program elements already accelerates distance calculation significantly, there are more efficient metrics possible on linear genetic programs.

One step toward a more efficient distance calculation between two effective programs is to give up the order of operators and to compare only the numbers of each operator type. Then program distance may be reduced to the Manhattan distance between two pattern vectors v and w of equal length n (n = size of operator set). Each vector position v_i represents the frequency of an operator type in the genetic program corresponding to v. The *Manhattan distance* is measured along axes at right angles and simply calculates the sum of absolute differences between equal vector positions, i.e., $\delta_{man}(v,w) = \sum_{i=1}^{n} |v_i - w_i|$. This requires runtime $O(n)$ only while n is much smaller here than for the edit distance. In other words, computation costs are constant here ($O(1)$) in terms of the maximum program length. Although the accuracy of this structural distance is definitely lower than the edit distance it has proven to be sufficient for an explicit control of diversity.

Another, more efficient distance metric than edit distance is applicable for controlling step sizes of (effective) instruction mutation. If a certain program position is varied, it calculates how many of the depending *previous* instructions in program (including the mutation point) have changed their effectiveness status. This is exactly the *Hamming distance* between the status flags and takes time $O(n)$ only with n is the maximum program length here.

A more precise Hamming distance may also compare the operator sequences such that unequal operator position increase the distance by 1. In this way, total distance 0 occurs less frequently because more variations are registered. For instance, micro mutations of single operator identifiers are detected. Even if the distance calculated by this metric is almost identical to the edit distance (for instruction mutations) we stick to the latter here for consistency reason. Note that, in general, the efficiency of distance calculation is less important for controlling variation distance than for controlling diversity (see below).

8.3 Semantic Program Distance

The most obvious metric to evaluate the behavior of a genetic program is the fitness function $\mathcal{F}$. This usually calculates the distance of the predicted outputs $gp(\vec{i_k})$ returned by a program and the desired outputs given by n fitness cases, i.e., input-output examples $(\vec{i_k}, o_k)$. For example, in Equation 8.1 this is simply the Manhattan distance between the two output vectors.

$$\mathcal{F}(gp) = \sum_{k=1}^{n} |gp(\vec{i_k}) - o_k| \tag{8.1}$$

Correspondingly, the semantic differences between two genetic programs may be expressed by their relative *fitness distance* (Equation 8.2). In this case, the quality of solving the overall problem is considered.

$$\delta_{fit}(gp_1, gp_2) = |\mathcal{F}(gp_1) - \mathcal{F}(gp_2)| \tag{8.2}$$

Another possibility is to compare the outputs of two programs directly. The same distance metric as in the fitness function may be used for computing the distance between the output vectors of programs (see Equation 8.3). In the following this will be referred to as *output distance*. Note that the relative output distance between two programs is independent from their performance in terms of solving a prediction task. Actually, two programs may have a similar fitness while their output behavior differs significantly, e.g., different subsets of the training data may be approximated with a different accuracy.

$$\delta_{out}(gp_1, gp_2) = \sum_{k=1}^{n} |gp_1(\vec{i_k}) - gp_2(\vec{i_k})| \tag{8.3}$$

Analogously, for discrete problems like classifications where the fitness function calculates a classification error, i.e., the number of wrongly classified examples, a *Boolean output distance* is defined as follows:

$$\delta_{boolout}(gp_1, gp_2) = \sum_{\substack{class(gp_1(\vec{i_k})) \neq class(gp_2(\vec{i_k})) \\ k=1,..,n}} 1 \tag{8.4}$$

Function *class* in Equation 8.4 hides the classification method that maps the continuous program outputs to discrete class identifiers.

8.4 Control of Diversity

In GP the *diversity* Δ of a population may be defined as the average distance of n randomly selected pairs of programs using a distance metric δ (see Equation 8.5).

$$\Delta = \frac{1}{n} \sum_{i=1}^{n} \delta(gp_{1i}, gp_{2i}) \tag{8.5}$$

The *genotype diversity* (or *structural diversity*) of programs is measured by means of a structural distance metric. Since we apply the edit distance between effective programs we refer to the *effective diversity*, accordingly.

We introduce the two-level tournament selection shown in Figure 8.1 for an explicit control of diversity. On the first level, individuals are selected by fitness. On the second level, the two individuals with *maximum* distance are chosen among *three fitter* individuals, i.e., tournament winners of the first round. While an absolute measure, such as fitness, may be compared between two individuals, selection by a relative measure, such as distance or diversity, necessarily requires a minimum of three individuals. In general, two from n individuals are selected with the greatest sum of distances to the $n-1$ other individuals.

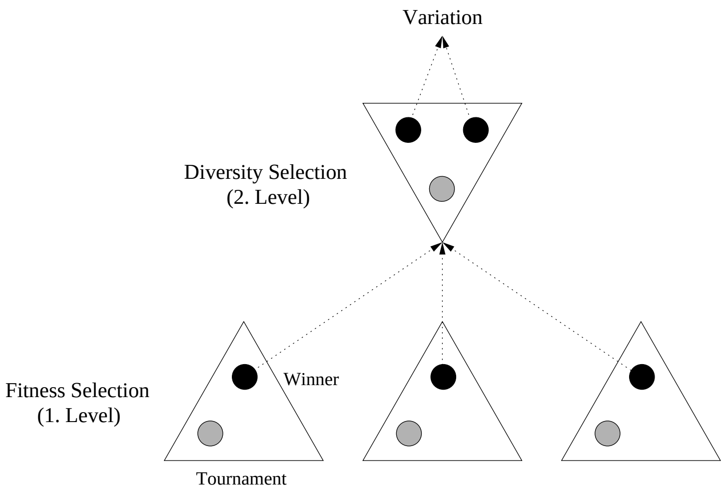

Figure 8.1: Two-level selection process.

Selection pressure on the first level depends on the *size* of fitness tournaments. Pressure of diversity selection on the second level is controlled by the *number* of these tournaments. Additionally, a selection rate controls how often diversity selection takes place at all and, thus, tunes the selection pressure on the second level more precisely.

The number of fitness calculations and the processing time, respectively, do not increase with the number of (first-level) tournaments if fitness of all individuals is saved and is updated only after variation. Only diversity selection itself becomes more computationally expensive the more individuals participate in it. Because n selected individuals require $\binom{n}{2}$ distance calculations an efficient distance metric is important here.

The two-level tournament selection constitutes a multi-objective selection method that finds individuals that are fitter *and* more diverse in relation to others. One advantage over applying fitness selection *or* diversity selection independently from each other on the *same* level is that the proportion of fitness selections is not reduced. Moreover, selecting individuals only by diversity for a certain probability does not result in more different directions among *better* solutions in the population. Dittrich *et al.* [27] report on a spontaneous formation of groups when selecting the most distant of three individuals that are represented by single real numbers.

Selection for a linear combination of both objectives, fitness and diversity, as this is often practiced with fitness and length (parsimony pressure), would require an appropriate weighting. This, however, is rather difficult to find. Another problem is that fitness and diversity still have the same priority. With the two-level selection, instead, fitness selection is not only decoupled from diversity selection but has always a higher priority.

An explicit control of effective diversity increases the average distance of individuals. Graphically, the population spreads more widely over the fitness landscape (see Section 5.4). Thus, there is a lower probability that the evolutionary process gets stuck in a local minimum and more different search directions may be explored in parallel.

While increasing the effective distance between programs in population affects the diversity of solutions, the absolute distance meassures a more general diversity including the noneffective code. Selection for absolute distance has also been practiced but found to improve results less (undocumented). Apart from the fact that this is more time-consuming it confirms that the absolute distance measures the effective program distance only very imprecise (see Section 8.2).

Increasing the average distance between programs by diversity selection has the side-effect of accelerating the growth of (effective) program length. In order to avoid that this may influence results, we select for the effective edit distance δ_{eff} minus the distance in effective length, i.e., $\delta_{eff}(gp_1, gp_2) - |l_{eff}(gp_1) - l_{eff}(gp_2)|$. This is possible because both edit distance and length distance operate on instruction (operator) level here. By doing so, a difference in length is no longer rewarded directly during selection. To further reduce the influence of code growth one might select (additionally) for the *relative effective distance* given by Equation 8.6. Note here that the size of the longest pattern string (effective program) determines the maximum effective distance.

$$\delta_{releff} = \frac{\delta_{eff}(gp_1, gp_2)}{\max(l_{eff}(gp_1), l_{eff}(gp_2))} \tag{8.6}$$

The diversity level can be *lowered*, too, by a selection for *minimum* distance. This might have a positive influence if population diversity is already quite high, e.g., because of a low fitness selection pressure or a low reproduction rate. In this case, especially crossover might profit from a reduction of diversity such that variation step sizes become indirectly smaller. In our experiments, however, selection for minimum distance resulted in the opposite (negative) effect as selection for maximum distance (undocumented).

Controlling *phenotype diversity* by a selection for a maximum *semantic distance* of individuals has been practiced by comparison. Semantic diversity is controlled by using the output distance defined in Section 8.3. A selection for maximum output distance may be implemented efficiently in both calculation time and memory usage, if only the outputs of individuals are saved that participate in the current tournament(s).

Selection for fitness distance has been found less suitable, instead. Note that both program fitness and program outputs are related to an absolute optimum. The relative output distance between programs, however, measures semantic differences more precisely. Increasing the relative fitness distance, instead, necessarily increases the diversity of fitness values in the population which promotes worse solutions. Moreover, selection by fitness distance has almost no effect on problems that implicate a rather narrow and discrete fitness distribution.

8.5 Control of Variation Step Size

One problem of GP is that already smallest variations of the symbolic program structure may affect program behavior heavily. In linear GP these variations especially include the exchange of registers. Several instructions that precede a varied instruction in a program may become effective or noneffective respectively. In this way, such mutations may not only affect the fitness, i.e., program semantics, but the flow of data in linear genetic programs that represents a directed acyclic graph (see Chapter 3). Even if bigger variations of program behavior are less likely with smaller structural variation steps, this effect is rather undesirable.

An *implicit control* of structural variation distance has been practiced in Chapter 5 by imposing respective restrictions on different types of variation operators. However, genetic operations – at least if changing a single variation point only (see Section 5.10.5) – may only guarantee for the *absolute* program structure that a certain maximum step size is not exceeded. Variation steps on the *effective* code, instead, may still be much bigger though these appear with a lower probability.

A major concern of this chapter is an *explicit control* of the *effective* variation distance. The variation of a parent program is repeated until its effective distance to the offspring falls below a *maximum threshold*. Therefore, the structural distance between parent and offspring is measured explicitly by applying the effective distance metric as defined above.

In the following extract of a linear program commented instructions are noneffective if we assume that the output is held in register `r[0]` at the end of execution. The program status on the right represents the result of applying an effective micro mutation to instruction number 8 (from the top). The first operand register `r[3]` is exchanged by register `r[2]`. As a consequence, 5 preceding (formerly non-effective) instructions become effective which corresponds to an effective mutation distance of 5.

```
void gp(r)                                     void gp(r)
  double r[5];                                   double r[5];
{                                              {
   ...                                            ...
// r[4] = r[2] * r[4];                         // r[4] = r[2] * r[4];
   r[4] = r[2] / r[0];                            r[4] = r[2] / r[0];
// r[0] = r[3] - 1;                               r[0] = r[3] - 1;
// r[1] = r[2] * r[4];                            r[1] = r[2] * r[4];
// r[1] = r[0] + r[1];                            r[1] = r[0] + r[1];
// r[0] = r[3] - 5;                               r[0] = r[3] - 5;
// r[2] = pow(r[1], r[0]);                        r[2] = pow(r[1], r[0]);
   r[2] = r[3] - r[4];             ->             r[2] = r[2] - r[4];
   r[4] = r[2] - 1;                               r[4] = r[2] - 1;
   r[0] = r[4] * r[3];                            r[0] = r[4] * r[3];
// r[4] = pow(r[0], 2);                        // r[4] = pow(r[0], 2);
// r[1] = r[0] / r[3];                         // r[1] = r[0] / r[3];
   r[3] = r[2] + r[3];                            r[3] = r[2] + r[3];
   r[4] = r[2] / 7;                               r[4] = r[2] / 7;
// r[2] = r[2] * r[4];                         // r[2] = r[2] * r[4];
   r[0] = r[0] + r[4];                            r[0] = r[0] + r[4];
   r[0] = r[0] - r[3];                            r[0] = r[0] - r[3];
}                                              }
```

Example 8.2: Change of effective code after effective register mutation (in line 8).

Since identical exchanges of instruction elements – including registers, operators, and constants – are avoided explicitly, operator mutations will always change the operator sequence. But operator mutations may induce a variation distance that is larger than 1, too, if the new operator requires a different number of parameters than the former operator. As a result, single registers may be either inserted or deleted *within* the particular instruction. Preceding instructions in program that depend on such a register operand may change their effectiveness status then by being reactivated or deactivated.

Besides restricting the maximum size of variation steps, we tested a *minimum threshold* as well. If small variation steps are avoided or, at least, reduced in frequency, evolutionary progress might be accelerated. Unfortunately, even smallest stuctural step sizes may already induce relatively large semantic step sizes. Our experimental results will show in Section 8.7 that the lowest maximum threshold that restricts effective step sizes to a minimum produces the best results.

Using an explicit control of the *fitness* distance between parent and offspring, instead, requires an additional fitness calculation after each iterated variation and can become computationally expensive, especially if a larger number of fitness cases is involved. By comparison, a structural distance like edit distance has to be re-calculated only once after

Problem	*sinpoly*	*iris*	*even8parity*
Problem type	Approximation	Classification	Boolean function
Problem function	$sin(x) \times x + 5$	real-world data set	even8parity
Input range	$[-5, 5]$	$[0, 8)$	$\{0, 1\}$
Output range	$[0, 7)$	$\{0, 1, 2\}$	$\{0, 1\}$
Number of inputs	1	4	8
Number of outputs	1	1	1
Number of registers	1+4	4+2	8+0
Number of examples	100	150	256
Fitness function	SSE	CE	SE
Number of generations	500	500	250
Instruction set	$\{+, -, \times, /, x^y\}$	$\{+, -, \times, /, if >, if \leq\}$	$\{\wedge, \vee, \neg, if\}$
Set of constants	$\{1, .., 9\}$	$\{1, .., 9\}$	$\{0, 1\}$

Table 8.1: Problem-specific parameter settings

each iteration while its computational costs do not directly depend on the number of fitness cases. It is also difficult to find appropriate maximum thresholds for the fitness distance because those are usually problem-specific. Finally, it is not sensible to restrict *positive* fitness changes (fitness improvement) at all.

8.6 Experimental Setup

All techniques discussed above have been tested with three benchmark problems including an approximation, a classification, and a Boolean problem. Table 8.1 summarizes problem attributes and problem-specific parameter adjustments of our LGP system.

The first problem is referred to as *sinpoly* in the following and denotes an approximation of the sine polynomial $sin(x) \times x + 5$ by non-trigonomical functions. Thus, given the facts that the maximum length of genetic programs is limited and that the *sine* function is defined by an infinite Taylor-series the optimum cannot be found. Besides the input register – that is identical to the output register – there are four additional calculation registers used with this problem. Recall that this additional program memory is important in linear GP, especially if the number of inputs is low by problem definition. With only one register the calculation potential is very restricted in most problem cases. Fitness is the *sum of square errors* (SSE). 100 fitness cases have been selected uniformly distributed over input range $[-5, 5]$.

The second problem *iris* is a popular classification data set that originates from the *UCI Machine Learning Repository* [15]. The real-world data contains 3 classes of 50 instances each, where each class refers to a type of iris plant. Fitness equals the *classification error* (CE), i.e. the number of wrongly classified inputs. A program output $gp(\vec{i_k})$ is considered as correct for an input vector $\vec{i_k}$ if the distance to the desired class identifier $o_k \in \{0, 1, 2\}$ is smaller than 0.1, i.e., $|gp(\vec{i_k}) - o_k| < 0.1$. Note that solution finding would be easier if this error threshold is extended to the maximum (0.5 here).

Finally, we have tested a parity function of dimension eight (*even8parity*). This function computes 1 if the number of set input bits is even, otherwise the output is 0. The Boolean branch in the instruction set is essential for a high number of successful runs with this problem. The number of wrong output bits, i.e., the sum of output errors (SE), defines the fitness.

Parameter	Setting
Population size	2000
Fitness tournament size	4
Maximum program length	200
Initial program length	10
Reproduction	100%
Micro mutation	25%
Macro mutation	75%
Instruction deletion	33%
Instruction insertion	67%
Crossover	75%

Table 8.2: General parameter settings.

More general configurations of our linear GP system are given in Table 8.2. Exactly one genetic operator is selected at a time to vary an individual program. Either linear crossover (cross, see Section 5.7.1) or (effective) instruction mutations ((eff)mut, see Section 5.10.4) are used as macro operator, but not in the same run. The absolute step size of macro mutations is minimum (1 instruction). Instead, an explicit bias (B1)) guarantees a sufficient growth of programs here (see Section 5.8).

8.7 Experimental Results

8.7.1 Distance Distribution and Correlation

First of all, we demonstrate experimentally that there is a causal connection between the structural distance and the semantic distance (fitness distance) of linear genetic programs when applying the edit distance metric on sequences of effective instruction operators as defined in Section 8.2. Causality forms a necessary precondition for the success of evolutionary algorithms. Even if already small modifications of the program structure may result in almost arbitrary changes in program behavior, smaller variations of the genotype should lead to smaller variations of the phenotype for a higher probability (see also Section 5.4.

In the first experiment distances of 2000 pairs of *randomly* selected individuals have been registered in each generation. Figures 8.2 to 8.4 visualize the resulting relation of (effective) program distance and fitness distance together with the corresponding distributions of program distances. In case of all test problems there is a clear positive correlation between program distance and fitness distance for the majority of measured distances. In principle, similar phenomena are observed here with the crossover-based and the mutation-based variant of linear GP.

In a second experiment that is relevant in this context we investigate the structural *variation* distance, i.e., the distance between parent and child or, more precisely, the distance of a modified individual from its original state. Figures 8.5 to 8.7 demonstrate a positive correlation between program distance and fitness distance, i.e, causality, for all tested combinations of problem and genetic operator. That is, shorter variation distances on code level induce shorter variation distances on fitness level, on average. The respective distributions of variation distances confirms this to be true for almost all measured distances.

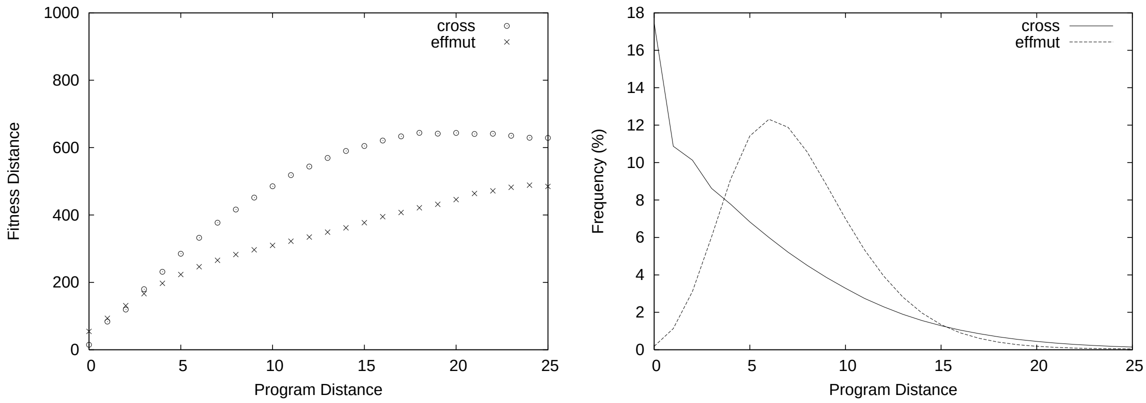

Figure 8.2: *sinpoly*: Relation of program distance and fitness distance (left) and distribution of program distances (right) in crossover runs (cross) and in runs using effective mutations (effmut). Average figures over 100 runs.

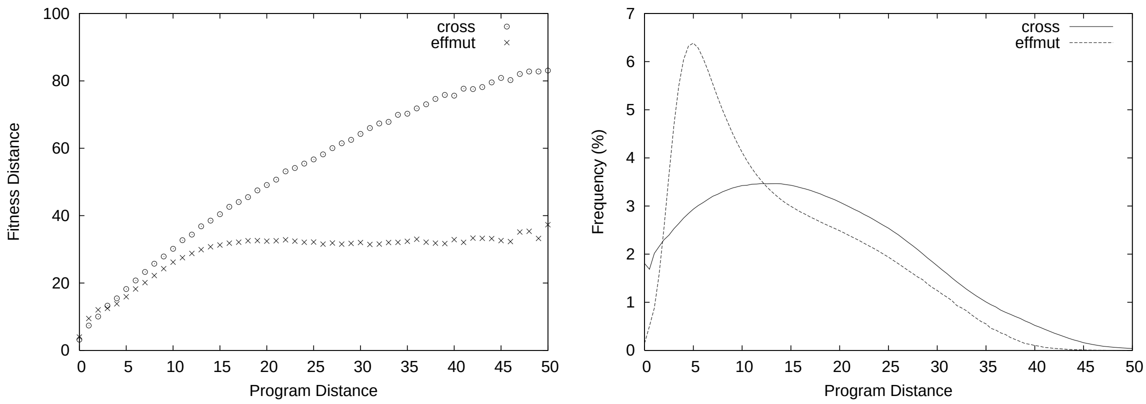

Figure 8.3: *iris*: Relation of program distance and fitness distance (left) and distribution of program distances (right).

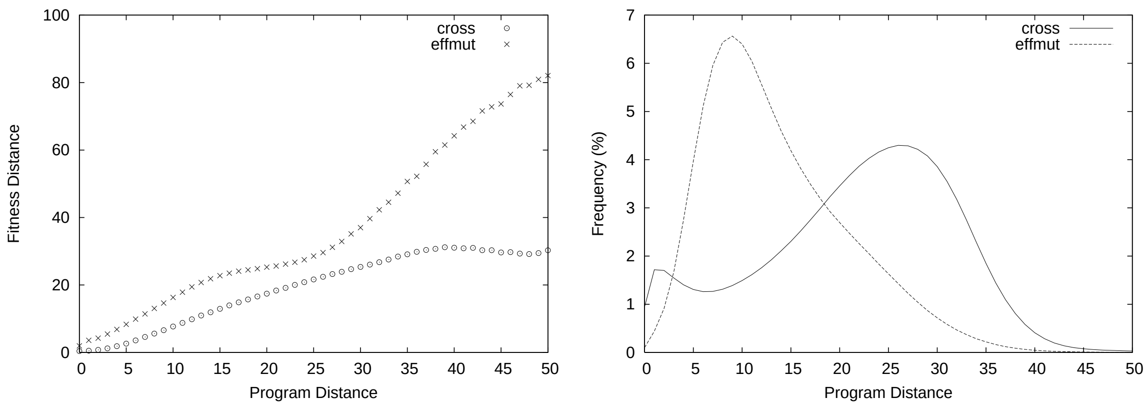

Figure 8.4: *even8parity*: Relation of program distance and fitness distance (left) and distribution of program distances (right).

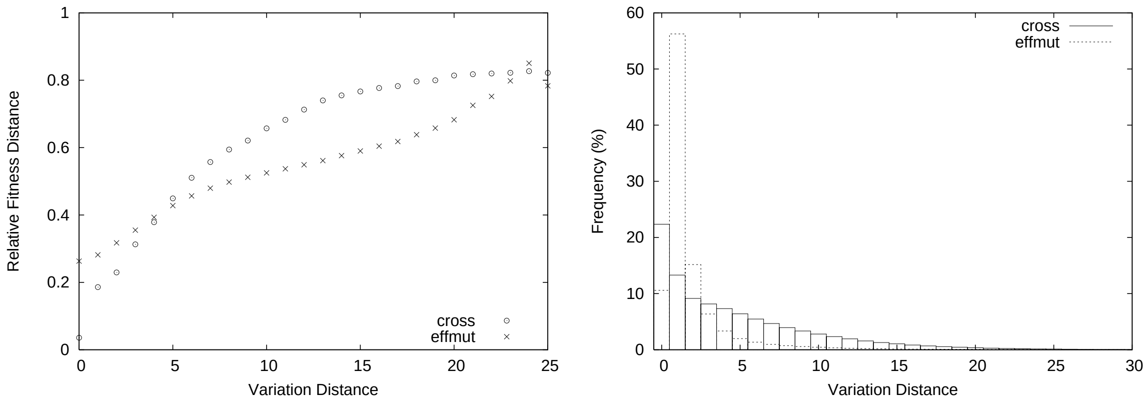

Figure 8.5: *sinpoly*: Relation of variation distance and relative fitness distance (left) and distribution of variation distances (right) in crossover runs (cross) and in runs using effective mutations (effmut). Average figures over 100 runs.

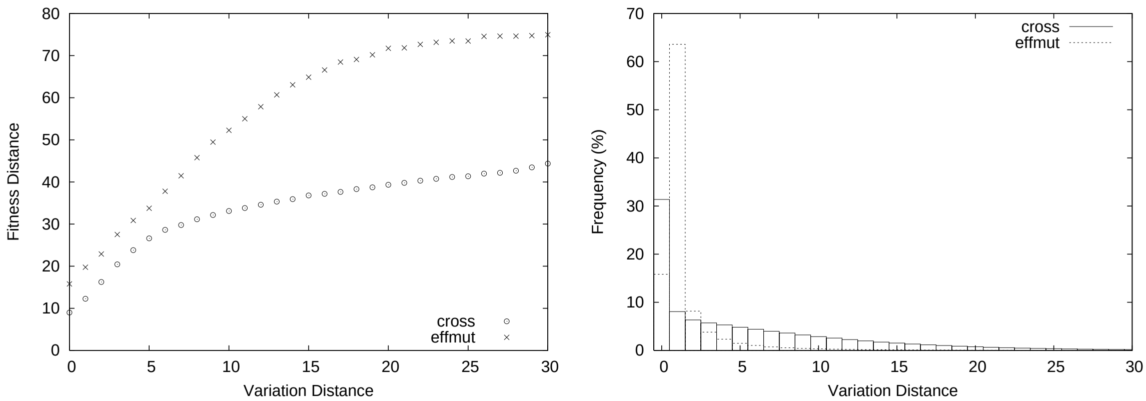

Figure 8.6: *iris*: Relation of variation distance and fitness distance (left) and distance of variation distribution (right).

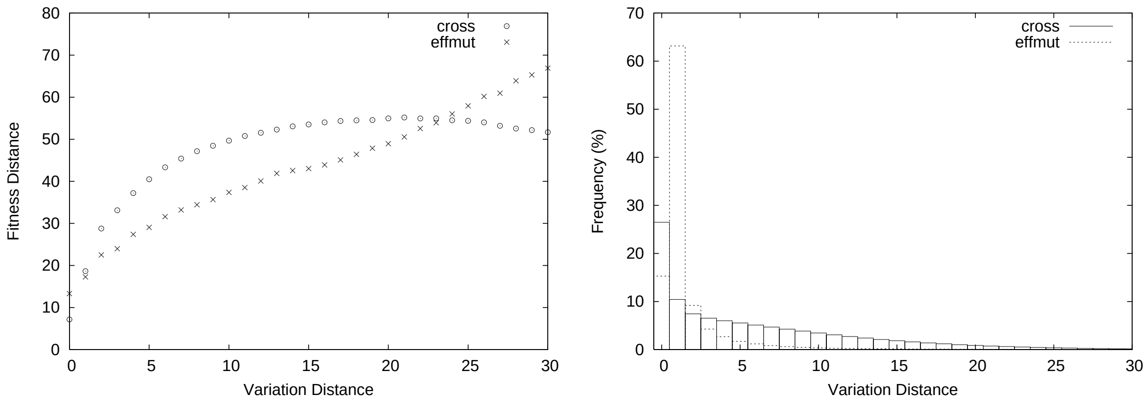

Figure 8.7: *even8parity*: Relation of variation distance and fitness distance (left) and distance of variation distribution (right).

While, in general, variation distances occur the more frequently the shorter they are, the distribution of crossover distances is wider than the distribution of distances induced by (effective) mutations.

Interestingly, small structural step sizes on the effective code still induce relatively large semantic step sizes, on average. This is more noticeable for effective mutations than for crossover. We will see in Section 8.7.7 that even if the effective step size is permanently minimum (1 for macro mutations) evolutionary progress is not decelerated. Since the functional representation of programs describes a rather narrow graph (see Section 3.3) already small changes may affect many data flow paths simultaneously.

The distribution range of distances is significantly smaller than in the first experiment, as might have been expected. That means the structural distance between parent and child is smaller, on average, than between two arbitrary individuals (or between two parents). This is an important property of evolutionary algorithms in general to work efficiently.

In crossover runs a high amount of operations results in effective distance 0, especially with the two discrete problems *iris* and *even8parity*. The reason is the high rate of structural introns (see Section 3.2) that occurs with crossover. Moreover, the 25 percent micro mutations used in all configurations will be most likely noneffective and, thus, produce effective step size 0. Recall, however, that not all (but only most) variations that induce distance 0 are necessarily noneffective, too, since our code-selective distance metric does not register all changes to the effective code (see Section 8.2).

As introduced in Section 5.10.4 effective (macro and micro) mutations definitely vary the effective code of programs. Effective distance 0 is mostly caused by effective *micro* mutations, especially those that affect a single register or constant. Since identical exchanges of such basic elements are avoided explicitly, operator mutations will always change the operator sequence. But not all substitutions of registers in effective instructions change the effectiveness of instructions and, thus, the sequence of operators, too.

Furthermore, distance distributions in Figures 8.5 to 8.7 show that almost two thirds of all effective mutations result in effective distance 1. Interestingly, even though macro mutations that insert or delete full effective instructions are applied in the majority of cases, effective distances larger than 1 occur for less than one third only. That means the effectiveness of other (preceding) instructions (except for the mutated one) changes for a relatively low probability.

8.7.2 Development of Effective Step Size

The 3D plot in Figure 8.8 demonstrates exemplarily for the *iris* problem how the distribution of effective step sizes develops over a run when using the effective mutation operator (effmut). The distribution is changing such that step sizes 1 and 0 occur more frequently while for larger step sizes the opposite is true. That is, after about 100 generations changes are caused almost exclusively at the mutation point rather than by deactivation of depending effective code. Deactivations are mostly responsible for larger effective distances. Reactivation of (structurally) noneffective code, instead, is much less likely because the proportion of this code remains less with the applied operator (see Section 5.11.1).

It appears that evolution develops effective program structures which are less fragile against stronger variation. We found that the effectiveness of an instruction is often guaranteed by more than one (succeeding) instruction. As demonstrated in Figure 8.9, the average effectiveness degree or dependence degree (see Section 3.4) of a program instruction grows continuously during a run. On the functional level this may be interpreted in such a way that the number of connections increases between nodes of the effective graph

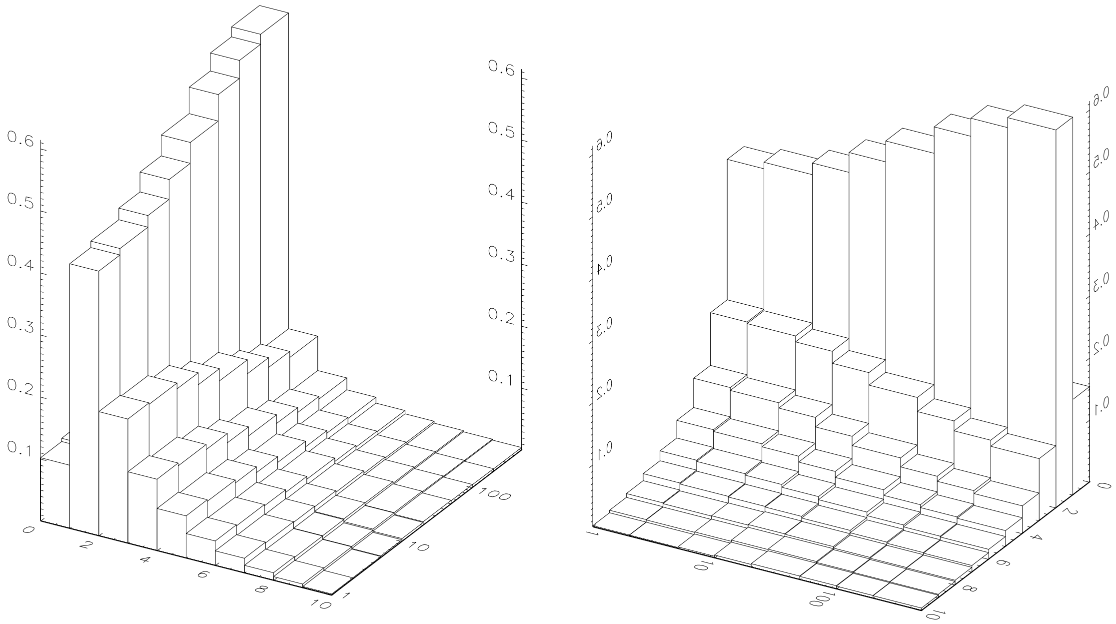

Figure 8.8: *iris*: Development of the frequency distribution of effective step sizes over 500 generations when using effective mutations (effmut). Step sizes range from 0 to 9 here. Frequency of step size 1 increases clearly over a run. Right figure same as left figure but rotated by 90 degrees.

component. Thus, the graph-shaped structure allows the effective code to protect itself against larger disconnections (deactivations). Smaller step sizes on the effective program structure will result in offsprings with a potentially higher fitness. This is true no matter whether this *self-protection effect* is an implicit evolutionary phenomenon or a consequence of the increasing power and complexity of solutions. In general, reducing the probability of deactivations by multiple node connections offers a fundamental advantage over tree programs where each node is connected to the root by only one edge (cmp. Section 7.5).

When investigating the evolution of effective step sizes it has to be considered that this depends on the evolution of (effective) program length, too. The larger programs become the larger step sizes are possible, in principle. Although programs grow over a run, the frequency of step sizes that are larger than 1 decreases in Figure 8.8 when a distance range of 0 to 9 is observed. Variation distances significantly larger than 10 instructions do not occur at the beginning of a run due to a small initial program length (see Section 8.6). But even if the *maximum* step size increases continuously with the program length in the course of a run, the proportion of all distances larger than 10 comprises about 2 percent only. Nevertheless, such events have an influence when calculating the *average* step size.

Figure 8.9 demonstrates that the average variation distance depends on the number of (calculation) registers. While smaller register numbers lead to a slightly decreasing or constant average effective step size, larger numbers lead to an increase. Such a behavior may be explained by the effectiveness degree of instructions again that turns out to be lower if more registers are available (see also Section 6.1). Then deactivations of code become more likely and affect larger parts of code. Nevertheless, the average step size remains relatively small even for large numbers of registers. It is important to note that the development of step sizes as shown in Figure 8.8 (for 2 calculation registers) is similar for different numbers of registers. The frequencies of step sizes 1 and 0 increase during

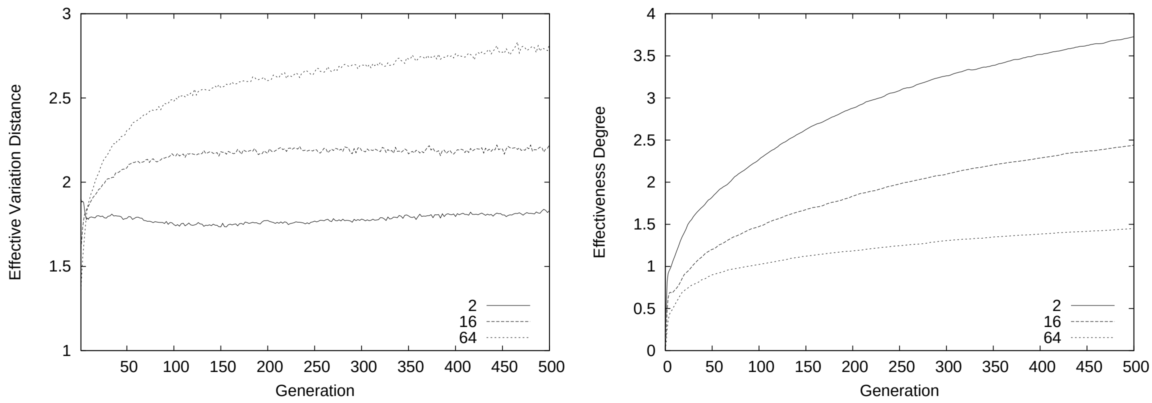

Figure 8.9: *iris*: Development of the average effective mutation step size (left) and the average degree of effectiveness (right) over the generations for different numbers of calculation registers (2, 16, 64) using effective mutations (effmut). Average figures over 50 runs.

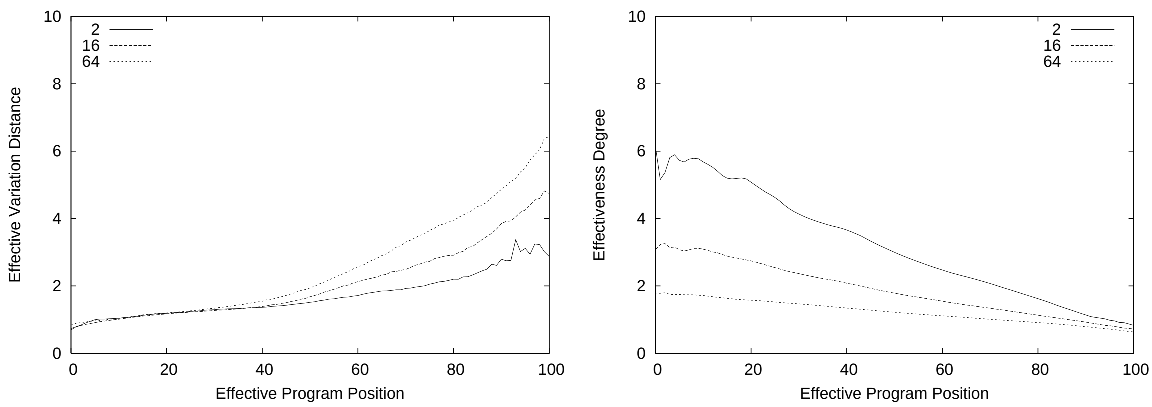

Figure 8.10: *iris*: Development of the effective mutation step size (left) and the degree of effectiveness (right) over the effective program positions. Position 0 holds the first instruction of a program. Average figures over 50 runs.

a run while the frequencies of step sizes 2 to 9 decrease. These two basic tendencies are only slightly understated if more registers are used in programs.

Larger step sizes do not result simply from larger programs here. Neither the size of effective code nor the size of noneffective code are significantly different for larger register numbers (undocumented). Moreover, the number of *effective* registers influences the effective step size and the self-protection effect, i.e., the decreasing proportion of larger effective step sizes over a run, only indirectly. As defined in Section 3.4, the number of registers that are effective at a certain program position reflects approximately the width of the corresponding program graph. Since a higher absolute number of registers involves wider but not larger program graphs the number of connections (dependence degree) decreases necessarily between the instruction nodes.

Figure 8.10 compares the effective step size and the effectiveness degree for different (effective) program positions. At the beginning of a program step sizes are similarly small for different register numbers. This part usually features the highest effectiveness, especially if the number of registers is small. Towards the end of a program the effectiveness decreases while the effective step size increases. Larger step sizes are correlated with higher register numbers here even if the effectiveness is similarly small. As noted earlier, the effective step size does not only depend on the effectiveness of the mutated instruction but also on the effectiveness of the preceding (depending) instructions in a program. Such developments follow from the graph-structured data flow in linear (effective) programs (see Section 5.11.6). Recall that the last effective instruction represents the root of the (effective) graph. Instruction nodes closer to the root have less connections and are, therefore, less protected against disconnections.

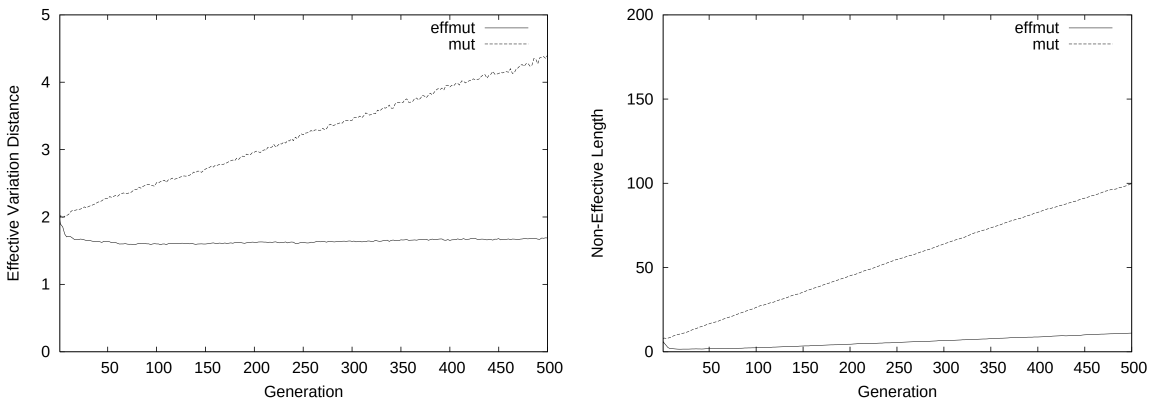

Figure 8.11: *iris*: Development of the average effective step size (left) and the number of noneffective instructions (right) for effective mutations (effmut) and free mutations (mut). Noneffective variations not regarded. Effective step size increases proportionally to the amount of noneffective code. The number of calculation registers is 2. Average figures over 100 runs.

When using random instruction mutations (mut) the amount of noneffective instructions in programs increases continuously during a run while it remains mostly constant with the effective mutation operator (see Figure 8.11). The number of effective instructions is even smaller here than this has been found with effmut (not shown). The resulting higher proportion of noneffective code leads to more noneffective variations (distance 0) and, thus, to a smaller average effective step size. But if only the effective variations (most distances larger than 0) are included, there is a linear increase in average step size (see Figure 8.11). Apparently, the increasing number of noneffective instructions increases the probability for reactivations. (As documented above, the effective step size on the effective code decreases, even though the effective code grows.) It has to be noted, however, that the increase in step size is still small compared to the increase in noneffective code. From this we may conclude that also the dependence between intron instructions increases over a run. The self-protection effect is however weaker here than for the effective code. Actually, noneffective instructions may be much more loosely connected in programs over register dependences, since they are not directly influenced by fitness selection. The reader may recall that this code can form several disconnected graph components on the functional level (see Section 3.3). This experiment identifies larger effective step sizes as a second reason, besides a higher rate of noneffective variations, why free instruction mutations perform worse than effective mutations.

Finally, we compare the development of effective step size for linear crossover (cross) in Figure 8.12. In contrast to the results found with instructions mutations, the step size *decreases* with a larger number of registers, even though the average effectiveness degree

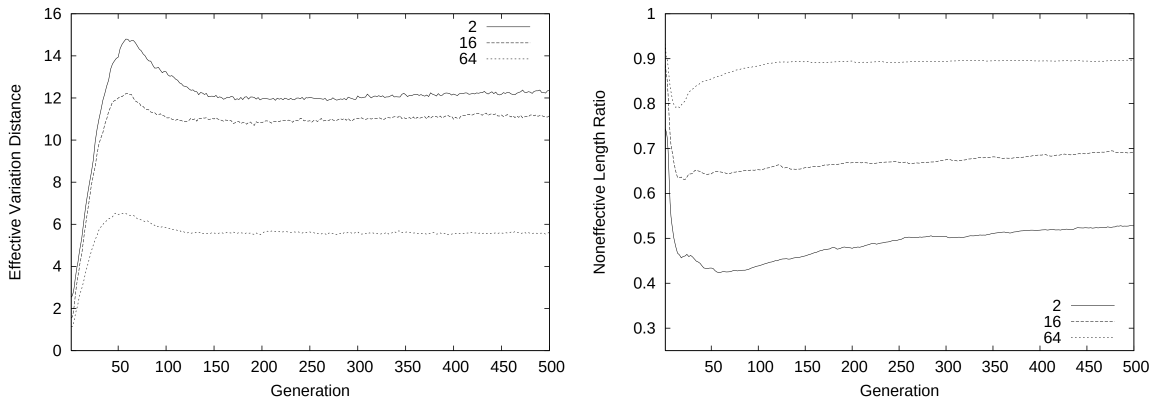

Figure 8.12: *iris*: Development of the average effective step size (left) and the proportion of noneffective length (right) over the generations for different numbers of calculation registers (2, 16, 64) using linear crossover (cross). Higher proportion of noneffective code leads to smaller effective step sizes. Average figures over 50 runs.

remains similar to that in Figure 8.9. This is true because a higher number of registers implies a higher proportion of noneffective code when using segment variations (see Section 6.1). As already noted, the proportion of noneffective instructions in a program may act as a second implicit protection mechanism that reduces the effective step size, besides the described self-protection effect. This is true at least for variations that comprise more than one instruction. Then a higher robustness of effective code seems to be less relevant for a reduction of effective step sizes than a higher rate of noneffective code. However, the former might be responsible for the small difference in effective step size between configurations with 2 and 16 calculation registers.

8.7.3 Structural Diversity Selection

For the three test problems introduced in Section 8.6, Table 8.3 shows average error rates obtained with and without selecting for structural diversity. Different selection pressures have been tested. For the minimum number of fitness tournaments (three) that are necessary for a diversity selection on the second level (see Section 8.4) we used selection probabilities 50 percent and 100 percent. Higher selection pressures are induced by increasing the number of tournaments (up to four or eight here).

The application of diversity selection is demonstrated with a population-dependent crossover-based approach and a mutation-based approach which is more independent from the diversity of the genetic material. It is conspicuous that in all three test cases linear GP works significantly better by using (effective) mutations instead of crossover. In Chapters 5 and Chapter 7 we have already demonstrated that the linear program representation, in particular, is more suitable for being developed by small mutations only, especially if those are directed towards effective instructions.

For each problem and both forms of variation the performance increases continuously by the influence of diversity selection in Table 8.3. The highest selection pressure that has been tested results in a twofold or higher improvement of prediction error. To achieve this, problem *sinpoly* requires a stronger pressure with crossover than the two discrete problems.

			sinpoly		*iris*		*even8parity*	
Variation	Selection		SSE		CE		SE	
	%	#T	*mean*	(± *std.*)	*mean*	(± *std.*)	*mean*	(± *std.*)
cross	0	2	3.01	(0.35)	2.11	(0.10)	58	(3.4)
	50	3	2.89	(0.34)	1.42	(0.08)	35	(2.4)
	100	3	2.77	(0.34)	1.17	(0.07)	27	(2.2)
	100	4	1.96	(0.22)	**1.09**	(0.07)	**19**	(1.8)
	100	8	**0.69**	(0.06)		—		—
effmut	0	2	0.45	(0.04)	0.84	(0.06)	15	(1.2)
	50	3	0.43	(0.03)	0.63	(0.05)	12	(1.0)
	100	3	0.30	(0.02)	0.60	(0.05)	10	(1.1)
	100	4	0.23	(0.02)	**0.33**	(0.04)	**7**	(0.8)
	100	8	**0.17**	(0.01)		—		—

Table 8.3: Second-level selection for *structural* diversity with different selection pressures. Selection pressure controlled by selection probability and number of fitness tournaments (T). Average error over 200 runs. Statistical standard error in parenthesis.

8.7.4 Development of Effective Diversity

In Section 8.4 the (structural) diversity of a population has been defined as the average effective distance between two randomly selected individuals. Figures 8.13 to 8.15 illustrate the development of diversity during runs for different selection pressures and different variation operators. The higher the selection pressure is adjusted the higher is the diversity. Interestingly, the average (effective) program distance does not drop even if diversity selection is not applied. Instead of a premature loss of diversity we observe an inherent increase of structural diversity during runs with linear GP. This is true even with the applied 100 percent reproduction and a selection pressure of four individuals per tournament. While the effective diversity increases with crossover until a certain level and stays rather constant then, the increase with effective mutations is more linear.

Such a behavior results partly from the variable-length representation in genetic programming. The longer effective programs develop during a run the larger effective distances are possible. The growth of effective code is restricted earlier with crossover by the maximum size limit than with effective mutations due to the much higher proportion of noneffective code that occurs with this operator – approximately 50 to 60 percent in the experiments conducted here. Nevertheless, by the influence of distance selection the average (effective) program length has been found to increase only slightly compared to the average program distance.

When using (macro) mutations a high degree of innovation is introduced continuously into the population. This may lead to a higher diversity of effective code than occurs with crossover (see Figures 8.13 to 8.15) in consideration of the fact that the average effective length is about the same here for crossover and effective mutations in the final generation.

The stronger it is selected for diversity, however, the more diversity is gaining ground in crossover runs. Apparently, there is a stronger influence of diversity selection with crossover than with mutations. Compared to mutation the success of recombination depends more heavily on the composition (diversity) of the genetic material in the population. The more different two recombined solutions are, the higher is the expected innovation of their offspring.

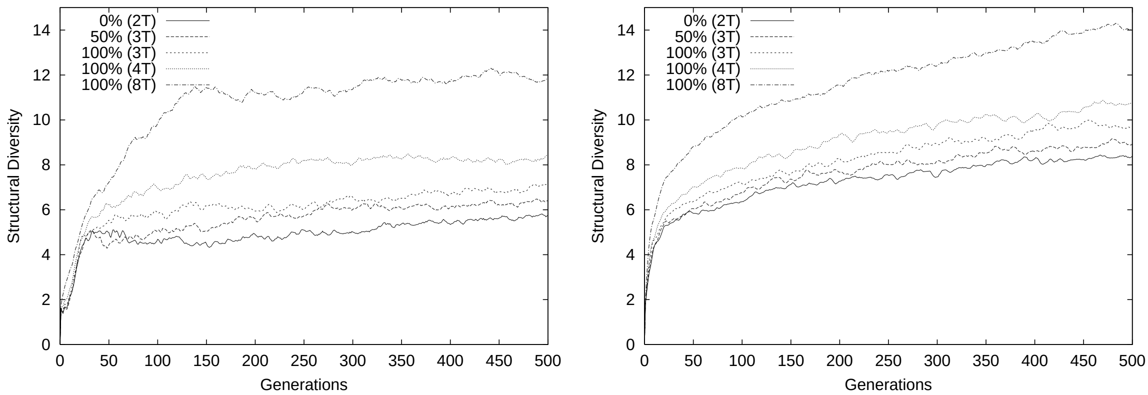

Figure 8.13: *sinpoly*: Structural diversity (average effective distance) with different selection pressures. Selection pressure controlled by selection probability and number of fitness tournaments (T). Average figures over 100 runs. Macro variation by sf cross (left) or effmut (right).

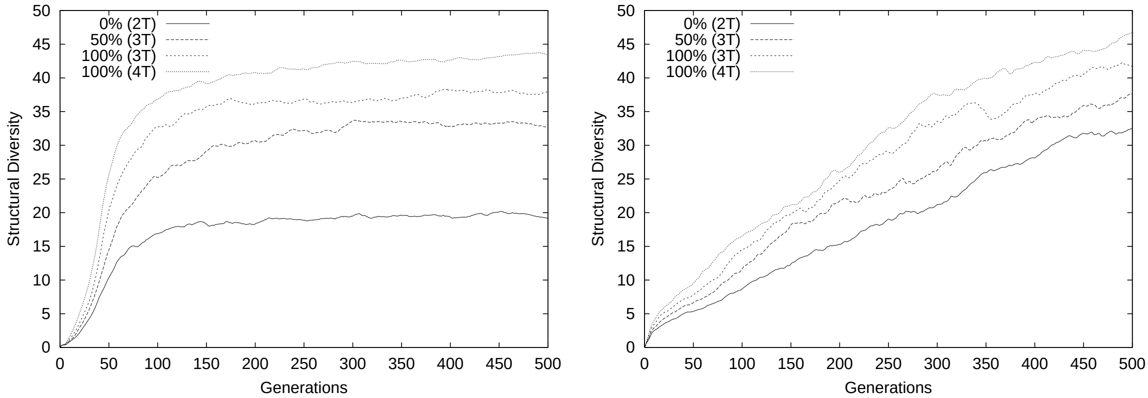

Figure 8.14: *iris*: Structural diversity with diversity selection and different selection pressures.

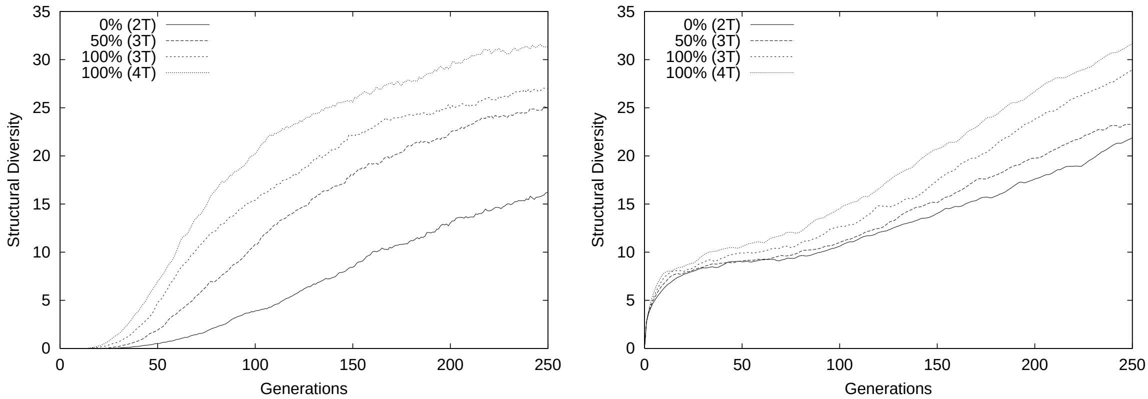

Figure 8.15: *even8parity*: Structural diversity with diversity selection and different selection pressures.

We also tested a diversity selection that uses the relative effective distance metric from Equation 8.6, exemplarily for the *iris* problem. (Similar results have been found with *even8parity* and *sinpoly* in principle.) Figure 8.16 compares the development of this *normalized effective diversity* with and without diversity selection. Apart from an early drop of diversity during the first 50 generations there is no further decrease in later generations. Actually, both forms of variation, linear crossover and effective mutation, maintain the diversity over a run already implicitly, i.e., without an explicit distance control. Note that crossover is applied for 100 percent here. For crossover the reason may be the free choice of crossover points which do not have to be the same for both parents in (linear) GP in contrast to other disciplines of evolutionary algorithms. As a result, even two identical parents may produce different offsprings. Another reason might be the large unrestricted step size of crossover. Finally, the high amount of noneffective code may contribute to the diversity of effective code with this operator.

The normalized diversity may even increase again in the course of a run, when using effective mutations and/or diversity selection. But at the end it levels off at a certain more-or-less constant value. Since the growth of effective code is hardly affected by the diversity selection here (see Figure 8.17) the influence of differently long patterns on the

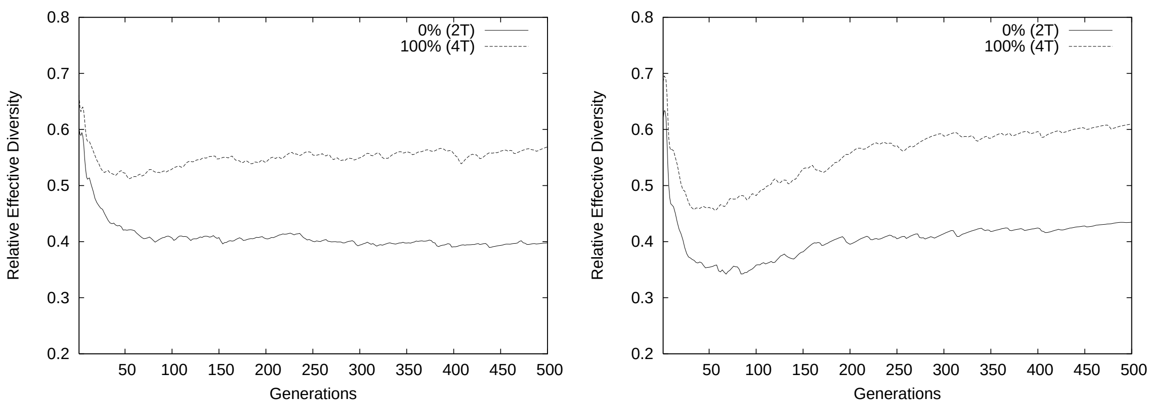

Figure 8.16: *iris*: Normalized effective diversity (average relative distance) with and without diversity selection. Selection pressure controlled by selection probability and number of fitness tournaments (T). Average figures over 100 runs. Macro variation by 100 percent cross (left) or effmut (right).

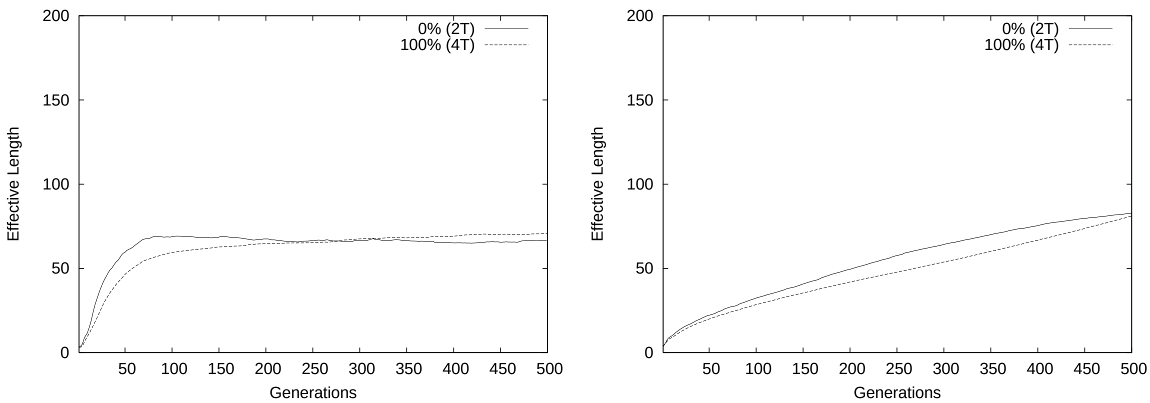

Figure 8.17: *iris*: Average effective program length with and without diversity selection. Difference in program lengths negligibly small compared to difference in diversity.

distance calculation can be neglected.

8.7.5 Semantic Diversity Selection

The computational overhead of a structural distance control has been found affordable for linear genetic programs, especially if the order of instructions is not regarded (see Section 8.2). In order to justify its usage more generally we test a semantic diversity selection for comparison. Semantic diversity is defined here as the average *output* distance of two individuals that have been randomly selected from the population (see Section 8.3). For each problem the same distance metric has been used as in the corresponding fitness function (see Table 8.1).

			sinpoly		*iris*		*even8parity*	
Variation	Selection		SSE		CE		SE	
	%	#T	*mean*	(± *std.*)	*mean*	(± *std.*)	*mean*	(± *std.*)
cross	0	2	3.01	(0.35)	2.11	(0.10)	58	(3.4)
	50	3	2.40	(0.22)	1.82	(0.09)	40	(2.5)
	100	3	3.51	(0.36)	1.62	(0.08)	46	(3.1)
	100	4	3.42	(0.33)	1.80	(0.09)	42	(2.8)
effmut	0	2	0.45	(0.04)	0.84	(0.06)	15	(1.2)
	50	3	0.33	(0.02)	0.77	(0.06)	13	(1.2)
	100	3	0.43	(0.03)	0.68	(0.05)	12	(1.1)
	100	4	0.49	(0.05)	0.42	(0.05)	9	(0.9)

Table 8.4: Second-level selection for *semantic* diversity with different selection pressures. Selection pressure controlled by selection probability and number of fitness tournaments (T). Average error over 200 runs. Statistical standard error in parenthesis.

When comparing results in Table 8.4 with results in Table 8.3 it follows that semantic diversity selection, in general, has a much smaller effect on the prediction quality than a selection for structural diversity. Especially the continuous problem *sinpoly* could not be solved more successfully by semantic diversity selection. For the two discrete problems we observe a significant influence only on runs with effective mutations.

One explanation is that, in contrast to program structure, program semantics is related to a unique optimum. For the program outputs this is the set of desired outputs given by the fitness cases. Hence, the number of possibly different output patterns reduces the closer fitness approaches the optimum (0). Compared with this the diversity of program structure is much more independent from fitness.

8.7.6 Diversity and Fitness Progress

Another interesting observation can be made when comparing the convergence of best fitness and population diversity over a *single* run. The fitness of the currently best individual reflects the progress of the evolutionary search.

First of all, there is no continuous increase of the average effective distance as one might expect from the average results over multiple runs (see Figures 8.13 to 8.15). The development of structural diversity in Figures 8.19 and 8.20 is interrupted by sudden rapid drops (*diversity waves*). Simultaneously, periods of fast fitness convergence can be observed where the currently best individual is replaced once or a few times in a row. Code

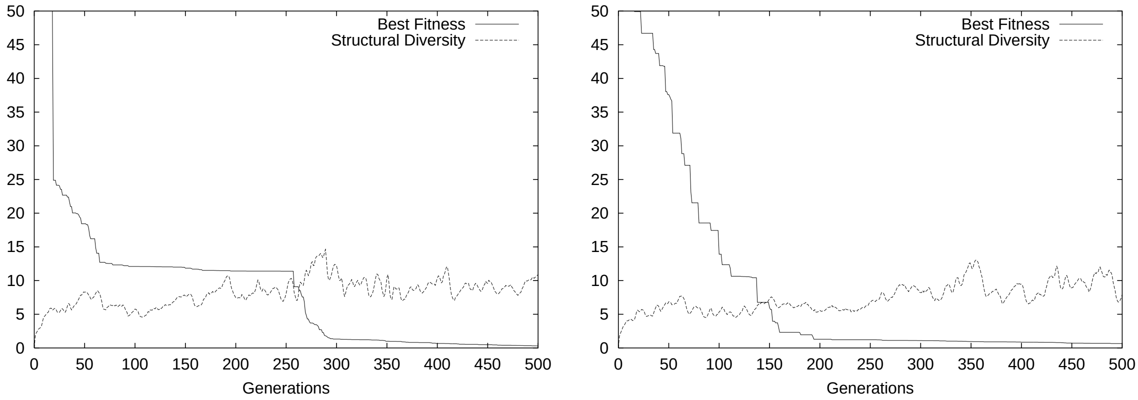

Figure 8.18: *sinpoly*: Development of best fitness and structural diversity. Two typical example runs.

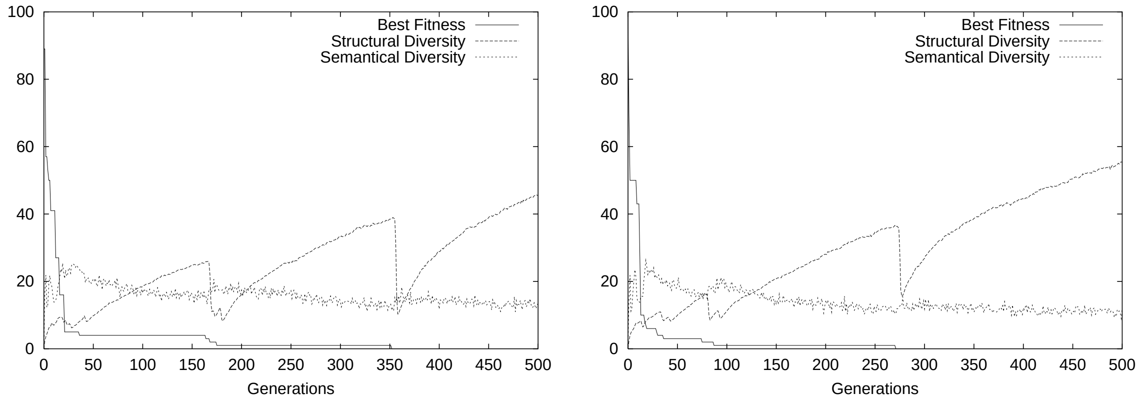

Figure 8.19: *iris*: Development of best fitness, structural diversity, and semantic diversity. Structural diversity grows during phases of fitness stagnation. Two typical example runs.

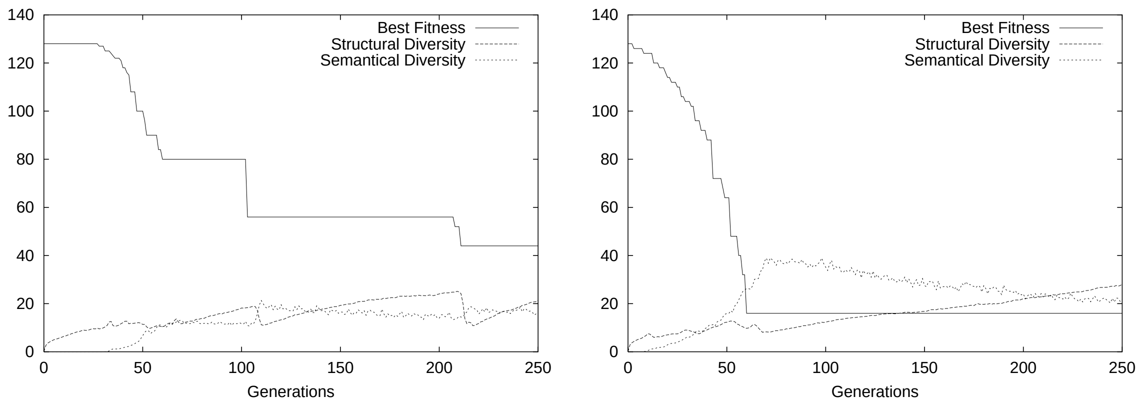

Figure 8.20: *even8parity*: Development of best fitness, structural diversity, and semantic diversity. Structural diversity grows during phases of fitness stagnation. Two typical example runs.

diversity decreases so quickly because a new best individual spreads in the population within a few generations via reproduction and variation. How quickly program diversity recovers after such an event depends on how many generations have elapsed so far. The higher a diversity level has been reached before the sharper is the increase. Typical example runs in Figures 8.19 and 8.20 demonstrate that structural diversity increases on fitness plateaus, i.e., during periods where the best fitness stagnates. During that time the population individuals spread over the fitness landscape and explore the search space of programs more widely. The achieved diversity level depends on both the duration of the stagnation period and the current number of generations. Comparable runs have been found with both kinds of macro variations.

A different behavior has been observed with the continuous problem (*sinpoly*). Structural diversity progresses wave-like, too, but with a higher frequency and a smaller amplitude (see Figure 8.18). A global correlation with the best fitness is less clear here already because the phases of fitness stagnation are shorter.

While structural diversity decreases quickly with the discrete problems when best fitness improves, a sudden increase of semantic diversity (average fitness distance here) can be observed. This phenomenon may be explained by a fast propagation of the new best fitness value in the population again by what semantically divers individuals are selected more frequently. During a period where best fitness stays constant the average fitness distance decreases again. The wider fitness range of the continuous problem, instead, allows stronger outliers. As a consequence, the average fitness distance develops too irregularly here (not printed).

It is important to note that the increase of structural diversity on fitness plateaus happens implicitly, that is without applying an explicit control of diversity. Using diversity selection increases the structural distance between individuals on fitness plateaus accordingly. Radical drops of diversity as a consequence of sudden accelerations of convergence speed, however, are just as possible as without diversity selection. This shows that an active increase of structural diversity does not slow down the *global* convergence of the best fitness over run. On the contrary, better prediction results have been observed with diversity selection in Table 8.3.

8.7.7 Control of Effective Mutation Step Size

In Section 8.5 we motivated to control the *effective* distance between parent and offspring explicitly. We will restrict ourselves to instruction mutations here. Recall that our distance metric regards instructions (operators) as smallest units. Correspondingly, variation is dominated by macro mutations with an *absolute* step size that is permanently minimum (1 instruction). In contrast to that, the effective step size may become significantly larger than 1 (see Section 8.7.1) and is altered by micro mutations, too (see Example 8.2).

In particular, we want to find out whether solution quality may be further improved by reducing the effective mutation distances probabilistically. Therefore, a program is mutated repeatedly until its distance to the offspring falls below a maximum threshold. Each time a mutation is not accepted its effect on the program is reversed while the choice of the mutation point is free in every iteration. In any case, iteration is stopped after a predefined maximum number of trails has been exceeded. The variation is executed once without restrictions then.

Table 8.5 compares average prediction errors for different maximum mutation distances. The maximum possible distance equals the maximum program length (200 instructions) and imposes no restrictions. Setting the maximum effective distance to 0 is not considered.

		sinpoly		*iris*		*even8parity*	
Variation	Maximum Distance	SSE		CE		SE	
		mean	(± *std.*)	*mean*	(± *std.*)	*mean*	(± *std.*)
effmut	—	0.46	(0.06)	0.90	(0.06)	16	(1.3)
	20	0.41	(0.05)	0.83	(0.06)	15	(1.2)
	10	0.35	(0.04)	0.72	(0.06)	13	(1.2)
	2	0.28	(0.03)	0.68	(0.05)	11	(1.1)
	1	**0.26**	(0.03)	**0.54**	(0.05)	**9**	(0.9)

Table 8.5: Maximum restriction of effective mutation distance. Average error over 200 runs. Statistical standard error in parenthesis.

This would not allow programs to grow even if it is possible that inserting an instruction does not change the effective distance. For all three benchmark problems best results are obtained with maximum effective distance 1. Thus, at most one instruction may change its effectiveness status, i.e., one node of the effective graph component is added or removed (see Section 3.3). It is interesting to note that in this case insertions or deletions of full instructions do not create non-effective code at all. For micro mutations this is true only for maximum step size 0.

Such a result is all the more interesting if we consider that a restriction of variation distance always implies a restriction in variation freedom, too. More specifically, certain modifications might not be executed at certain program positions because too many other instructions would be affected. It is important in this context to check the required number of iterations until a mutation gets accepted. On the one hand, the average number of iterations reveals how strongly the variation freedom is restricted. On the other hand, multiple recalculations of effective distance may produce computational costs that cannot be neglected here.

Variation	Maximum Distance	#Iterations *sinpoly*	*iris*	*even8parity*
effmut	—	1.00	1.00	1.00
	10	1.02	1.02	1.02
	5	1.06	1.05	1.05
	2	1.18	1.12	1.12
	1	1.37	1.18	1.20

Table 8.6: Average number of iterations until a maximum mutation distance is met.

As we can learn from Table 8.6, the average number of iterations until a maximum effective distance is met increases only slightly if the threshold is lowered. Not even one and a half iterations are necessary, on average, with the smallest distance. Besides, the maximum number of iterations (10 here) has hardly ever been exceeded. Both aspects, together with the results from Table 8.5 emphasize that freedom of variation is restricted only slightly and that computational overhead of this distance control is affordable.

It may be pointed out that these results correspond to the distribution of mutation distances in Figures 8.5 to 8.7 where about 20 to 40 percent of all measured step sizes are larger than 1. Hence, effective programs become increasingly robust against larger disruptions since this increases their survival probability. Two main reasons for this have been identified in Section 8.7.2. First, the effectiveness of an instruction depends on more than one succeeding instruction in a program, on average. This reduces the probability that deactivations of effective instructions increase the effective step sizes. Second, because of the low rate of noneffective instructions that has been found with effective mutations reactivation may hardly play any role here.

Even though the average effective step size has turned out to be small already implicitly, an explicit minimization leads to an even better performance. This is due to the fact that, on average, minimum step size on the (effective) program structure is still comparatively large on the semantic level (see Figures 8.5 to 8.7).

8.8 Alternative Selection Criteria

A two-level tournament selection may also be used for implementing complexity control, as we will see in Section 9.9.4. The separation of linear genetic programs into effective and noneffective instructions offers the possibility for a *selective* complexity selection. That means it may be selected specifically for the smallest *effective*, *non-effective*, or *absolute* program length.

Diversity selection and complexity selection may be applied in combination, too. Either a third selection level is added or both objectives are combined into a weighted sum for selection on the second level. In the latter case, selection priority for diversity and complexity may be more-or-less the same. In the first case, this may be achieved by using an independent selection probability for each level. Then selection for minimum length may happen on the second level while selection for maximum distance is skipped on the third level.

Besides a smaller length or a larger distance of programs there are other properties of linear genetic programs that may be selected for (see Chapter 3). For instance, one might want to select for a smaller or larger average number of effective registers in linear genetic programs. Like optimum program length, i.e, optimum number of nodes, optimum width of functional program structure may vary with problem definition. Another possible alternative might be to select for a higher effectiveness of instructions, i.e., for a higher connectivity of nodes. In doing so, programs are preferred whose effective code is protected best.

Finally, an active selection for more diverse individuals may also be used to reduce the population size significantly without leading to a decrease in performance. By maintaining the (same) level of diversity, a smaller population may still cover a wide area of the search space, even if less search points are examined simultaneously. Smaller population sizes mean less fitness evaluations per generation what may result in an enormous speedup, especially in time-critical applications.

Basically, the development of population diversity over a run is dependent on the following control parameters of an EA: *population size*, *fitness selection pressure* (*tournament size*), and *reproduction rate*. The structural distance metrics introduced here for linear GP allow a detailed analysis of such parameter influences. It remains to be a subject of future research, for instance, how strongly larger population sizes are correlated with higher diversity. In the experiments above these parameters have been configured with (constant) standard settings. Nonetheless, we experienced that diversity selection works with very different configurations, including smaller and larger population sizes. The only adaptation that might be necessary is a reconfiguration of selection pressure.

8.9 Conclusion

In this chapter we measured and controlled the diversity of effective programs and the effective step size of different variation operators explicitly for three different benchmark problems. We proposed different metrics to calculate structural or semantic distance between linear genetic programs. The following conclusions may be drawn:

(1) A clear positive correlation between structural distance and fitness distance of programs was demonstrated. In particular, measuring structural differences specifically between subcomponents of effective programs has been found to demonstrate causality of variation step sizes.

(2) An explicit control of code diversity was introduced in terms of a two-level selection process that selects for fitness on the first level and for diversity on the second level. Fitness selection always has higher priority with this multi-objective selection method. By increasing structural distance between effective programs (effective diversity) in the population performance improved significantly.

(3) The level of effective diversity has been found to stabilize early during a run even if crossover is applied exclusively. This level is directly determined by the applied selection pressure on diversity.

(4) Instruction mutations were introduced in Chapter 5 to cause minimum structural variations on linear genetic programs. Only one instruction was varied to let programs grow or shrink. In this chapter we tried to achieve this on the level of effective code, too. In particular, it turned out to be most successful if not more than one effective instruction in a program changes its effectiveness status through mutation. On the functional level only one node of the effective graph component may be added or removed. Thereby, the average number of iterated mutations that is necessary to comply with this condition was small.

(5) Effective mutation step sizes were measured much smaller than expected. Actually, effective program structures emerged that were quite robust against larger destructions (deactivations) in the course of evolution. An increasing degree of effectiveness of instructions was held responsible for this self-protection effect. In this way, multiple connections of instruction nodes (on the functional level) offer a fundamental advantage of linear programs over tree programs.

Chapter 9

Code Growth and Neutral Variations

Contents

This chapter brings together theories about neutral variations and code growth in genetic programming. In doing so, the importance of neutral variations for the growth of code is emphasized. Existing theories about code growth are verified for linear GP, in particular, and are partly reevaluated from another perspective.

In evolutionary computation neutral variations are argued to explore flat regions of the fitness landscape while non-neutral variations exploit regions with (positive or negative) gradient information. We investigate the influence of different variation effects on the growth of code and the prediction quality for different kinds of variation operators. It is a well-known fact, that a high proportion of neutral code (introns) in genetic programs may increase the probability for variations to become neutral. But which type of variation creates the intron code in the first place ? Especially if linear GP is applied with minimum mutation step sizes results show that neutral variations almost exclusively represent a cause of (and not only a result of) the emergence and the growth of intron code. The influence of non-neutral – especially destructive – variations on code growth has been found to be considerably smaller, by comparison, even if larger variation step sizes are applied.

Furthermore, different linear genetic operators are examined for an implicit length bias. In contrast to an explicit bias, an implicit bias does not result from the dynamics of the operator only, but requires the existence of a fitness pressure.

We close with some considerations about how code growth may be controlled in linear GP. Different ways are suggested including variation-based methods and selection-based methods. Both may be done specifically for the effective code and/or the noneffective code of linear genetic programs. In particular, it will be demonstrated that mutation on linear genetic programs influences code growth much less than recombination. This is the more true the less code growth is limited by other factors, like the maximum program size or the maximum step size.

9.1 Code Growth in GP

One characteristic of genetic programming is that the variable-length individuals grow in size. To a certain extent this growth is necessary to direct the evolutionary search into regions of the search space where sufficiently complex solutions with a high fitness are found. It is not recommended in general to initiate the evolutionary algorithm already with programs of a too large or even maximum size (as demonstrated in Section 6.6). If the initial complexity of programs is too high the population may be too inflexible to develop towards a region of the search space with highly fit programs.

However, by the influence of the variation operator – especially the variation step size – and other reasons that are discussed in this chapter genetic programs may grow too fast and too large such that the minimum size of programs required to solve the problem is exceeded significantly. As a result, finding a solution may become more difficult. This negative effect of code growth, i.e., that programs become larger than necessary without corresponding fitness improvements became known as the *bloat effect*. Code growth has been widely investigated in the GP literature [51, 2, 16, 65, 84, 54, 85, 55, 90, 12] (see below).

In general, a high complexity of GP programs causes an increase of evaluation time and reduces the flexibility of genetic operations in the course of the evolutionary process. Besides, unnecessarily large solutions are more difficult to analyze and may lead to a worse generalization performance [79].

Depending on the proportion of noneffective code that occurs with a certain combination of variation operators, the problem of longer processing time may be relaxed significantly in linear GP by removing structural introns from a genetic program each time before its fitness is calculated (see Section 3.2.1). Thus, only the (structurally) effective code causes relevant computational costs during program execution.

The length of a linear genetic program is measured as the number of instructions it holds. As already noted, the *absolute program length* and the *effective program length* are distinguished in linear GP. Correspondingly, we distinguish code growth concerning all instructions from the growth of (structurally) effective instructions only. This is referred to as *absolute growth* and *effective growth*, respectively.

9.2 Proposed Causes of Code Growth

Several theories have been proposed to explain the phenomenon of code bloat in genetic programming. Basically, three different causes of code growth are distinguished up to now that do not contradict each other and may coexist while each being capable of causing code growth for itself. Most theories explain the growth of intron code. In general, the minimally required complexity of a solution may be exceeded by incorporating intron code (may be removed without changing the program behavior) or by mathematically equivalent extensions (see Chapter 3). All causes require the existence of fitness information, i.e., may not hold on (completely) flat fitness landscapes. In this way, fitness may be regarded as a necessary precondition for code growth. Only the (semantically) effective program size directly depends on the fitness. At least to a certain extent, solutions have to increase their effective complexity to improve their fitness performance.

We assume for the following considerations that all variation operators are designed and configured such that they are not explicitly biased towards creating longer offsprings more frequently, at least not independently from the fitness selection.

9.2.1 Protection Theory

The *protection theory* [65, 16, 12, 90] argues that code growth and, in particular, the growth of introns occurs as a protection against the destructive effects of variation. The protection effect is sometimes explained by an increasing proportion of neutral variations (and a corresponding decrease of destructive variations) that results from a higher rate of intron code in programs. We will demonstrate below why such an explanation may not be a sufficient one. First, the rate of destructive variations is not necessarily decreasing during a run, especially if the variation step size is large, e.g., restricted only by the program size (see Section 9.8.4). Second, in this case programs may even grow without neutral and/or destructive variations (see Section 9.8.3). Finally, neutral variations reduce the number of variations that happen to the non-neutral code which may not always be advantageous.

A more general explanation for the protection effect and its influence on code growth may be found by regarding the structural step size of variations. In particular, this includes non-neutral variations, too. The destructive influence of a variation on the program structure strongly depends on its step size. If the maximum amount of code is large or even unrestricted that may be exchanged or deleted in one variation step (absolute step size), evolution may reduce the variation strength on the effective code (effective step size) by developing a higher proportion of introns in programs and, thus, in the varied subprograms. This phenomenon may occur when using crossover as well as subprogram mutations. In this way, the intron code controls the (*relative*) effective step size which depends on the ratio of effective and noneffective code in programs. Programs with a higher rate of noneffective code (and the same absolute length) produce fitter offsprings on average, i.e., offsprings with a higher survival probability. It is argued that code grows because such offsprings will be more likely reselected for reproduction and variation [65].

Nevertheless, it is true that a higher intron rate in programs may increase the probability for variations to become neutral, especially if the variation step size is small. This is not only valid for code deletions, but also for insertions. Note that in larger intron regions the number of effective registers may be supposed to be lower. In particular, the effective step size is zero for neutral variations while the survival probability of offsprings is definitely higher after neutral variations than after destructive variations.

9.2.2 Drift Theory

Another theory (*drift theory*) [54, 55] claims that code growth results from the structure of the search space or, more precisely, from the distribution of semantically identical solutions. The same phenotype function may be represented by many structurally different (genetic) programs. There are many more larger genotypes than there are smaller ones for a certain fitness value. This is caused by intron code or mathematically equivalent code extension. Therefore, the genetic operators will create longer offsprings for a higher probability that perform as well as their parents. Since the population programs represent a sample of the search space, longer solutions will be selected more frequently, too. Both will make the population to evolve in a random drift towards more complex regions of the search space.

This general drift theory may be criticize because it assumes that longer programs emerge due to a certain structure of the search space only. It has to be noted that not all programs of the search space are created equally likely and, thus, may be composed of an arbitrarily large amount of introns. This depends strongly on the applied variation operator and, in particular, on the variation step size (see discussion below). Only because genetic operators search in genotype space, the programs in the population do not have to become significantly larger than necessary, as demonstrated in Section 5.11 for the effective mutation approach. Hence, the part of the actual search space that is visited by a certain operator may be much smaller than the search space of all possible solutions.

9.2.3 Bias Theory

A third theory (*bias theory*) of code growth is based on the hypothesis of a removal bias in tree-based GP [86, 55, 90]. The change caused by removing a subtree can be expected the more destructive the bigger the subtree is. The effect of the replacing subtree on the fitness, instead, is independent from its size. As a results, the growing offspring from which the smaller subtree is removed (and in which the longer is inserted) will survive for a higher probability than the shrinking offspring.

It has to be noted, however, that the size of the exchanged subprograms may not be the only reason for code growth. The lower fitness of the parent individual from which the larger subtree is extracted may simply result from the fact, too, that the subtree root (crossover point) lies closer to the tree root, on average. In this region crossover is more likely destructive. Accordingly, the smaller subtree originates more likely from lower tree regions.

The removal bias theory presumes that there are no side-effects induced by the program functions in the problem environment. It is further important that both parents have the same size, on average, since the destructiveness of a removed subtree depends on the absolute size of program, too. Finally, this cause strongly relies on the fact that the variation operators only affect a single point in a program tree. We will see in Section 9.8 that such an implicit grow bias cannot be identified that clearly in linear GP.

9.3 Influence of Variation Step Size

The (maximum) step size of a variation operator determines the potential speed of code growth that is possible in one variation step but does not represent a direct cause. In general, we have to distinguish more-or-less *necessary preconditions* (indirect causes) for code growth from *driving forces* (direct causes) as introduced in the last section. A larger step size reduces the probability for neutral variations, but increases the probability that neutral code may directly emerge from non-neutral variations.

If we want to clearly identify a direct or indirect reason for code growth it is important to design the experiment in such a way that other causes are disabled as much as possible. The protection effect (see Section 9.2.1) may be at least significantly lower if the step size of variation operators is reduced to a minimum and if code is not exchanged. Both may be achieved in linear GP for the imperative program structure by mutations that insert *or* delete single random instructions only, as described in Section 5.10. (No code growth is possible by substitutions of single instructions only.) Then a protection effect may not occur in form of a reduction of effective step size, at least for all non-neutral variations that alter the program length. The only protection effect that is remaining may result from reducing the *proportion* of destructive variations in favor of neutral variations. This is possible by a higher intron rate in programs.

If the mutation step size is constantly one, intron instructions cannot be inserted or deleted *directly* along with a non-neutral variation, but only by a neutral variation. In particular, this allows destructive variations to be analyzed with only a minimum influence on the amount of intron code. Introns may only emerge indirectly from non-neutral variations by deactivation of depending instructions (apart from the mutation point). The larger the intron code has already developed the more likely this situation becomes. This is true for introns on the structural level and on the semantic level. With large or even unrestricted step sizes, instead, programs may grow quickly even by a small number of variations.

The high variability of the linear representation allows structural step sizes to be permanently minimum at each program position. Reasons for this are both the graph-based data flow and the existence of structural noneffective code in linear genetic programs (see Section 3.3). Due to stronger constraints of the tree representation, small macro variations are especially difficult in upper tree regions. If single tree nodes are tried to be deleted, for instance, only one of its subtrees may be reconnected while the others get lost (see also discussion in Chapter 7). Also due to structural constraints, introns hardly occur in nodes near the root but are concentrated near the leaves [90]. Probably, the number of effective nodes might be too restricted, otherwise.

A possible drift effect is reduced, too, because the difference between parent and offspring comprises only one instruction. By using minimum variation steps exclusively the evolutionary process will drift less quickly towards more complex regions of the search space. In particular, a drift of intron code is hardly possible by non-neutral variations then.

9.4 Neutral Variations

Most evolutionary computation approaches model the Darwinian process of natural selection and adaptation. In the Darwinian theory of evolution organisms adapt to their environment in such a way that mutations of the genotype spread in a population if they offer a fitness advantage. Natural selection is considered to be the dominating force for molecular evolution. In particular, the theory claims that most changes by mutations are expressed in fitness. Most mutations are believed to be destructive and to be sorted out of the population quickly by selection. That is, a mutation is only believed to survive over generation if it improves the fitness.

Contrary to this theory, Kimura's [48] neutral theory states that the majority of evolutionary changes on molecular level are due to neutral or nearly neutral mutations. The neutral theory does not deny the existence of natural selection but assumes that only a small proportion of changes happens adaptively, i.e., follows a fitness gradient. The bigger proportion of mutations is believed to stay silent on phenotype level, i.e., have no significant influence on survival or reproduction. Those neutral genes spread within populations by a random genetic drift which is considered to be a main force of evolution. The neutral theory is supported by recent experimental data [49].

In linear GP we discern two types of neutral variations. While *noneffective neutral variations* change the (structurally) noneffective code only, *effective neutral variations* change the effective code, too (see Section 5.1). The first type may be avoided if genetic operations are explicitly guaranteed to alter the effective code. In Chapter 5 neutral instruction mutations have been identified as a motor of evolutionary progress. Best results were obtained by increasing the proportion of effective neutral mutations actively.

Neutral variations do not provide any gradient information to the evolutionary algorithm. This reduces the probability for improving the fitness by a gradient descent (*exploitation*). Instead, neutral variations allow evolution to faster overcome plateaus of the fitness landscape. As a result, the fitness landscape may be explored more widely and searched more efficiently for potentially better suboptima (*exploration*). In doing so, neutral variations may be expected to prevent the evolutionary search from getting stuck in local suboptima.

When destructive variations dominate the evolutionary process, it is harder for an individual to improve step-by-step and to spread within the population. For a higher probability it will get worse with each mutation until it is replaced by a better individual. By neutral variations, instead, an individual may be altered without changing its ability to succeed in fitness selection. This offers evolution the possibility to develop solutions "silently", i.e., without exposing changes to fitness selection after each variation step. This intron code may become relevant when being reactivated later in the course of the evolutionary process (see Section 9.8.5). In principle intron manipulations may be carried out by non-neutral variations, too, if the variation step size is large enough. However, it is important to note that they will survive less likely since the vast majority of such variations is destructive.

Banzhaf [10] first emphasized the relevance of neutral variations in genetic programming when a search space of almost unconstrained genotypes (binary strings) is distinguished from a search space of constrained phenotypes (program trees) in genetic programming. A special genotype-phenotype mapping is applied to guaranteed the feasibility of phenotype solutions while the genetic operators may search in the genotype space without constraints.

Yu and Miller [103] demonstrated that neutral variations are advantageous after extra neutral code has been explicitly included into a graph representation of programs. A better performance was found for a Boolean problem (even-3-parity) if neutral mutations are allowed in a modified (1+4)EA, compared to accepting fitness improvements only.

It has to be noted, however, that the proportion of constructive variations is usually rather low in genetic programming why in the latter case only a very small proportion of variations may have an influence on the evolution of code. The authors do not compare their results with an approach that accepts destructive variations, too.

9.5 Conditional Reproduction

We use a steady state evolutionary algorithm (see Section 2.3) that applies tournament selection with a minimum of two participants per tournament. Variations happen on copies of the parent individuals that may either replace the originals in the populations (no reproduction) or the tournament losers (reproduction). When using tournament selection the reproduction rate determines the number of parent individuals that survive a variation step, i.e., that are taken over into the next "generation" of the steady-state population together with the offsprings. With such a local selection scheme, it is not recommended in general to restrict the reproduction rate significantly in genetic programming. Even if diversity is better preserved in a population if less individuals are overwritten, fitness convergence may be influenced rather negatively. This is not only true because better solutions may spread more slowly within the population but because these individuals get lost with a higher probability, especially if the proportion of destructive variations is high. In particular, the loss of a new best-fit individual becomes possible if reproduction is not strictly applied with tournament selection. Because of the high complexity of genetic programs and the comparatively low rate of constructive variations (improvements) during a GP run, information that has once been lost is hard to be regained in the following evolutionary process.

The question is now under which conditions reproduction may be skipped without risking to loose better solutions and when it is absolutely necessary. Obviously, after noneffective variations the effective code has not changed and is already completely reproduced through the offspring individual. In this case, the variation already includes a reproduction and additional copies of the parent individuals do not contribute to the preservation of information, but only to a loss of diversity. If reproduction happens after effective variations only, solution-relevant information cannot get lost while unnecessary reproductions of the effective program are avoided. This approach is referred to as *effective reproduction* and is another method to better preserve the effective diversity in the population, besides the diversity selection discussed in Chapter 8. It may, however, be applied only if not all variations are effective.

Noneffective variations, by definition, are always neutral in terms of a fitness change, but not vice versa. While noneffective variations preserve the effective solutions completely, skipping the reproduction step after neutral variations is more critical. If a neutral variation alters the (structurally) effective code the original solution code gets lost. Moreover, such variations may only be neutral in terms of the fitness cases, not in terms of all possible input data. This may reduce the generalization performance.

An omission of the reproduction step after destructive variations is even less motivated, since better individuals would be exchanged by worse. This necessarily must lead to worse results. Finally, reproduction after constructive variations should be retained, already because the probability of such events is rather low.

9.6 Conditional Variation

Besides the reproduction of the parent individuals, the integration of newly created individuals into the population (by replacing tournament losers) may be restricted so that offsprings are accepted only if they result from certain types of variation. Such a *conditional acceptance* of a variation implies automatically that the reproduction of parents is omitted, too, since the population remains unchanged. Otherwise, if reproduction would always take place, the parental information is doubled while overwriting existing information in the steady-state population.

9.7 Experimental Setup

The different experiments documented in this chapter are conducted with the four benchmark problems that have already been introduced in Section 5.8.1. Unless otherwise agreed the same system configuration is used here as in Section 5.8.2. Variants of this standard configuration will be described with the corresponding experiments in the following.

9.8 Experimental Results

9.8.1 Conditional Instruction Mutations

The experiments documented in Tables 9.1 to 9.4 investigate the influence of different variation effects on both the complexity of (effective) programs and the prediction performance. The average prediction error is calculated by the best solutions of 100 independent runs together with the statistical standard error. The absolute and the effective program length are averaged over all programs that are created during runs. (Figure 9.1 shows exemplarily the generational development of the average program length in the population.) Due to the small step size of mutations used here, the average length of best individuals develops almost identically (not documented). The proportion of effective code is given in percent while the remaining proportion comprises the structural introns. Additionally, we calculate the average proportions of constructive, neutral and noneffective variations among all variations during a run. The rates of destructive and effective variations are obvious then.

In the no∗ experiments of Tables 9.1 to 9.4 offsprings are not inserted into the population if they result from a certain type of variation. Additionally, the reproduction of the parent individuals is skipped. Simply put, the variation is canceled completely without affecting the state of the population. Note that this is different from the control of neutrality discussed in Section 5.10.7 where variations are repeated until they are neutral. Nevertheless, with all configurations the same number of variations (and evaluations) happens, i.e., the same number of new individuals (1000) defines a generation. Thus, non-accepted variations are still included in the calculation of the prediction error, the program lengths and the variation rates.

Standard instruction mutations (mut) are characterized by a balanced emergence of neutral operations and non-neutral operations, on the one hand, and effective operations and noneffective operations, on the other hand.

Destructive variations hardly contribute to the evolutionary progress here. For all test problems, the prediction error changes only slightly compared to the standard approach if offsprings from destructive variations are not accepted (nodestr). This is true even though

Experiment	SSE		Length			Variations (%)		
	mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
mut	3.5	0.5	140	60	43	0.8	54	52
nodestr	3.3	0.5	139	61	44	0.2	53	52
noneutr	1.6	0.1	**38**	**28**	72	7.5	37	34
nononeff	1.5	0.1	**41**	**30**	74	4.8	41	32
effrepro	**1.5**	0.2	126	50	40	3.3	60	52

Table 9.1: *mexican hat*: Conditional acceptance of mutation effects and conditional reproduction (mut, B1). Average results over 100 runs.

Experiment	SSE		Length			Variations (%)		
	mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
mut	6.5	0.3	78	32	41	0.5	63	63
nodestr	8.0	0.3	78	32	41	0.1	64	63
noneutr	6.0	0.3	**24**	**15**	63	6.3	48	47
nononeff	6.5	0.2	**25**	**16**	62	4.7	52	48
effrepro	**4.8**	0.3	56	25	44	4.1	61	58

Table 9.2: *distance*: Conditional acceptance of mutation effects and conditional reproduction (mut, B0). Average results over 100 runs.

about 50 percent of all variations are rejected and even if the rate of constructive variations decreases significantly, especially with the classification problems (in Tables 9.3 and 9.4). In contrast to that the rate of neutral variations remains more-or-less unaffected in this experiment. Obviously, the probability for selecting an individual, that performs worse than its parent, seems to be so low, on average, that it hardly makes any difference if this individual is copied into the population or not. Due to a low survival rate of these offsprings and due to the small mutation step size, destructive mutations almost do not have any influence on code growth here, too. Note again that intron instructions cannot be directly inserted by a *non*-neutral variation and *all* changes of a program are exposed to fitness selection.

The influence of neutral variations is in clear contrast to the influence of destructive variations. Obviously, the survival probability of offsprings is higher after a neutral (or a constructive) variation. This facilitates both a continuous further development of solutions and the growth of programs. In doing so, neutral variations explore plateaus of the fitness landscape by a random walk. It is an important result that both the absolute size and the effective size of programs are reduced most if we exclude neutral variation results from the population (noneutr).[1]

Noneffective neutral variations create or modify noneffective instructions, i.e., structural introns. Accordingly, we may assume that mostly *effective neutral* variations are responsible for the emergence of semantic introns – within the (structurally) effective part of program. Effective neutral variations (and semantic introns) are harder to induce if the fitness function is continuous and, thus, occur less frequently. This is reflected here with the two regression problems by similar rates of noneffective operations and neutral operations. For the discrete classification problems, instead, the proportion of neutral variations

[1] This is true here even if an explicit grow bias has been used with some problems (see Section 5.8).

Experiment	CE		Length			Variations (%)		
	mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
mut	13.6	0.6	128	64	50	0.3	50	42
nodestr	12.4	0.5	117	64	55	0.02	46	39
noneutr	20.0	0.6	**37**	**31**	82	5.0	32	20
nononeff	13.1	0.5	69	62	89	1.5	32	13
effrepro	**9.2**	0.4	117	83	71	1.1	45	25

Table 9.3: *spiral*: Conditional acceptance of mutation effects and conditional reproduction (mut, B1). Average results over 100 runs.

Experiment	CE		Length			Variations (%)		
	mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
mut	15.5	0.6	132	57	43	0.2	62	49
nodestr	16.4	0.7	124	53	43	0.03	62	49
noneutr	24.6	0.8	**34**	**28**	82	5.3	38	20
nononeff	12.9	0.7	80	71	88	1.0	45	13
effrepro	**12.4**	0.6	116	89	76	0.7	54	22

Table 9.4: *three chains*: Conditional acceptance of mutation effects and conditional reproduction (mut, B1). Average results over 100 runs.

has been found significantly larger than the proportion of noneffective variations which means a higher rate of effective neutral variations.

Additionally, the frequency of neutral variations on the effective code depends on the function set. Especially, branches create semantic introns easily while the resulting larger effective code indirectly increases the probability for effective (neutral) variations.

In the nononeff experiments noneffective variations are rejected, i.e., only effective variations are accepted. In contrast to the noneutr, this includes effective neutral variations, too. Semantic introns created by those variations may be responsible for the larger effective code that occurs with both classifications in nononeff runs. With the two regressions the effective size is half-reduced for both noneutr and nononeff because most neutral variations are noneffective here. If we would compare results after the same number of *effective* evaluations this approach more-or-less corresponds to the effmut operator that calculates effective mutations algorithmically.

In both noneutr and nononeff runs the rate of noneffective code is reduced significantly. As a result, the rates of neutral variations and noneffective variations are smaller here. This demonstrates that the intron code in programs does not only emerge mostly from neutral variations, but increases the probability for a neutral variation again.

We may conclude that neutral variations – in contrast to destructive variations – dominate code growth almost exclusively. Since mutation step sizes are small, constructive variations may only play a minor role for code growth already because of their low frequency. This is true even if the rate of constructions increases (together with the rate of destructions) when not accepting the result of neutral variations in the population (noneutr). One reason for this is the lower rate of structural and semantic introns. Moreover, non-neutral variations may hardly be responsible for an (unnecessarily) growth of code here because the variation step size is minimum. Then intron code cannot be directly created by such operations and *all* changes of a program are exposed to fitness selection.

As already noted in Section 9.1, the possibility to induce small structural mutations at each position of the linear representation is important for our results. Indirect creation of intron instruction by deactivations seems to play a minor role only. Note that due to changing register dependences noneffective (effective) instructions may be reactivated (deactivated) in a linear genetic program above the mutated instruction. Besides, an increasing robustness of the effective code lets deactivation of instructions occur less frequently in the course of a run (see Section 8.7.2).

When step sizes are larger, i.e., more than one instruction may be inserted per variation, as this occurs with crossover, programs may grow faster and by a smaller total number of variations. In particular, introns may be directly inserted by variations, too, that are not neutral as a whole.

Concerning the prediction quality the noneutr experiment has a small positive or no effect with the two approximation problems but a clear negative effect with the two classification problems. Contrary to this, the performance never drops in the nononeff experiment (compared to the baseline result). Apparently, fitness is not negatively affected if only noneffective neutral variations are excluded. Consequently, effective neutral variations may be supposed to be more relevant than noneffective neutral variations in general. This is not obvious, because all neutral changes may be reactivated later in (non-neutral) variations.

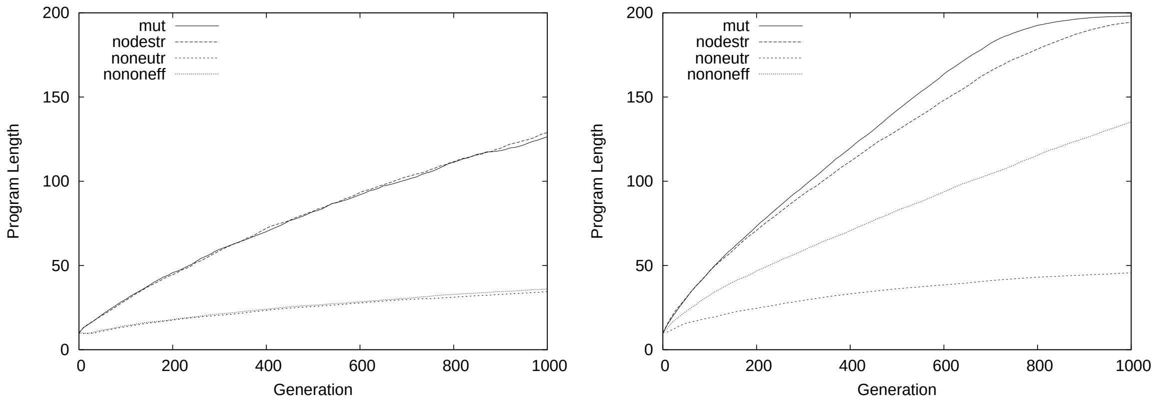

Figure 9.1: Development of average absolute program length for *distance* (left) and *three chains* (right) (similar for *mexican hat* and *spiral*). Code growth significantly reduced without neutral variation effects. Average figures over 100 runs.

We may not automatically conclude here that neutral variations are more essential for solving classifications only because those problems are discrete. At least small plateaus in fitness landscape also exist with problems whose output range is continuous. It has to be noted, that a better performance may also result from the fact that programs grow larger by neutral variations due to a step-by-step improvement of solutions. Depending on the problem definition, the configuration of the instruction set, and the observed number of generations, the optimum speed of code growth may be quite different. By making use of branches, that allow many special cases to be considered in a program, both classification problems profit less from a lower complexity of solutions than the two symbolic regressions.

9.8.2 Effective Reproduction

Reproduction after effective operations only (effrepro) is characterized by a clear gain in performance compared to the standard approach (mut) in Tables 9.1 to 9.4. Since the reproduction step is rather pointless if the effective code has not been altered (see Section 9.5), the diversity of solutions may be better maintained without. Recall that about 50 percent of all variations are noneffective with mut. This assumption is also confirmed by a higher average fitness and standard deviation that have been found with effrepro (not documented).

In contrast to nononeff, newly created individuals are always accepted and find their way into the population here. Interestingly, the average prediction error is smaller than or equal to the error obtained in nononeff runs. This is probably due the fact that the (effective) program size is less reduced here by a lower reproduction rate of parents than by a lower acceptance rate of their offsprings.

9.8.3 Conditional Segment Variations

Soule *et al.* [85] demonstrated for tree-based GP that code growth (especially of introns) remains significantly lower if only those offsprings are incorporated into the population that perform better than their parents. The authors hold the missing destructive crossover results directly responsible for this behavior. The researchers observed that the reduced complexity of programs is mostly due to a much lower rate of intron code – using a control problem where (semantic) intron code is partly easy to identify in program trees. The researchers also observed that the size of effective code is reduced in size.

While a direct influence of destructive variations on the growth of (intron) code is not doubted in principle here, it has to be noted, however, that not only destructive but also neutral variations are excluded from evolutionary progress in [85]. Moreover, the proportion of (the remaining) constructive variations is usually rather low in GP. It may be difficult to decide then whether the reduced program growth is not just the result of too few individuals that find their way into the population.

This section documents the influence of different variation effects on code growth when using unrestricted segment operators in linear GP – including two-segment recombination (crossover, cross) and one-segment mutations (onesegmut). In Tables 9.5 to 9.8 either destructive variations (nodestr), neutral variations (noneutr) or both (noneutr+nodestr) have been canceled in separate experiments. In doing so, both the reproduction of parents as well as the integration of offsprings into the population are skipped for the corresponding variation types.

Since variation step sizes comprise more than one instruction (structural and semantic) intron instructions may be inserted by both neutral and non-neutral variations, in principle. Here the segment length is restricted by the absolute program length only. In general, the more instructions may be inserted in one variation step the less variations are necessary to let programs bloat provided that there is at least one valid reason for code growth for the applied genetic operator(s).

As already documented in Section 5.9.2, smaller solution sizes occur in general when using (one-)segment mutations instead of recombination in Tables 9.6 and 9.8. It will be argued in Section 9.9.2 that this is a result of the fact that randomly created segments restrict the formation and propagation of introns in the population. Similar to the results found with instruction mutations in Section 9.8.1 code growth is hardly affected here if destructions are not accepted (nodestr). As noted above, the fitness of an offspring might be comparatively low within the population after a destructive variation. Therefore, it

Experiment	SSE		Length			Variations (%)		
	mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
cross	15.4	1.5	180	67	37	4.9	26	22
nodestr	12.4	1.4	177	68	38	0.5	23	22
noneutr	9.9	1.2	170	70	42	10.9	21	18
noneutr+nodestr	3.3	0.4	122	53	43	2.8	19	17

Table 9.5: *mexican hat*: Conditional acceptance of variation effects using crossover (cross). Average results over 100 runs after 1000 generations.

Experiment	SSE		Length			Variations (%)		
	mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
onesegmut	4.2	0.5	92	38	42	4.6	26	21
nodestr	5.3	0.6	99	43	43	0.2	20	19
noneutr	2.9	0.2	96	43	44	10.4	23	18
noneutr+nodestr	3.2	0.2	75	36	48	2.0	20	19

Table 9.6: *mexican hat*: Conditional acceptance of variation effects using one-segment mutation (onesegmut). Average results over 100 runs after 1000 generations.

is rather unlikely for a program solution to be processed and to grow in a sequence of destructive operations (without being overwritten).

In contrast to Section 9.8.1, however, programs grow here even if neutral offsprings do not get into the population (noneutr). A significantly smaller complexity has been found only for the *spiral* classification when using one-segment mutations. Without neutral variation effects the performance decreases for this problem. Instead, fitness improves significantly

Experiment	SSE		Length			Variations (%)		
	mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
cross	26.1	0.7	185	102	55	3.6	23	14
nodestr	25.0	0.7	184	103	56	0.1	21	15
noneutr	27.6	0.6	174	106	61	8.7	25	12
noneutr+nodestr	26.1	0.5	101	57	56	1.1	23	13

Table 9.7: *spiral*: Conditional acceptance of variation effects using crossover (cross). Average results over 100 runs after 1000 generations.

Experiment	CE		Length			Variations (%)		
	mean	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
onesegmut	21.2	0.6	126	65	51	2.4	27	19
nodestr	18.0	0.7	125	66	53	0.04	23	18
noneutr	27.8	0.6	63	36	56	7.2	29	17
noneutr+nodestr	31.4	0.5	37	21	59	0.7	25	19

Table 9.8: *spiral*: Conditional acceptance of variation effects using one-segment mutation (onesegmut). Average results over 100 runs after 1000 generations.

for the *mexican hat* problem. Additionally, in both cases the rate of constructive variations is more than doubled compared to the standard approach. It is important to note that constructive operations are responsible for the growth of noneffective and effective code here since the variation step size is unrestricted. The difference in average fitness with and without using neutral variations cannot result from a difference in solution size, at least for *mexican hat*, as this may be true for the corresponding test series with instruction mutations in Section 9.8.1.

If both neutral and destructive changes are rejected (noneutr+nodestr) the evolutionary progress and code growth are controlled by constructive variations exclusively. Since the rate of constructions is even lower here than in normal runs hardly any new individuals get into the population. Average code size is limited significantly only with the *spiral* problem (see Table 9.8).

The maximum size limitation lets the average program length be more similar in the crossover experiments (Tables 9.5 and 9.7). Only Figure 9.2 reveals significant differences if the maximum limitation is chosen so large (1000 instructions) that it may not affect the development of program lengths until about generation 200 with *mexican hat* and until about generation 125 with *spiral*. In general, one can see that code growth is more reduced without neutral variation effects than without destructive effects, even if destructions occur three times more frequently. On the *mexican hat* problem destructive variations even do not seem to have any influence at all. It also becomes clear here that code growth is much more restricted if neither destructive nor neutral crossover effects are accepted. Then the comparatively low number of constructive effects is not sufficient to let programs bloat even though arbitrarily large segments are used.

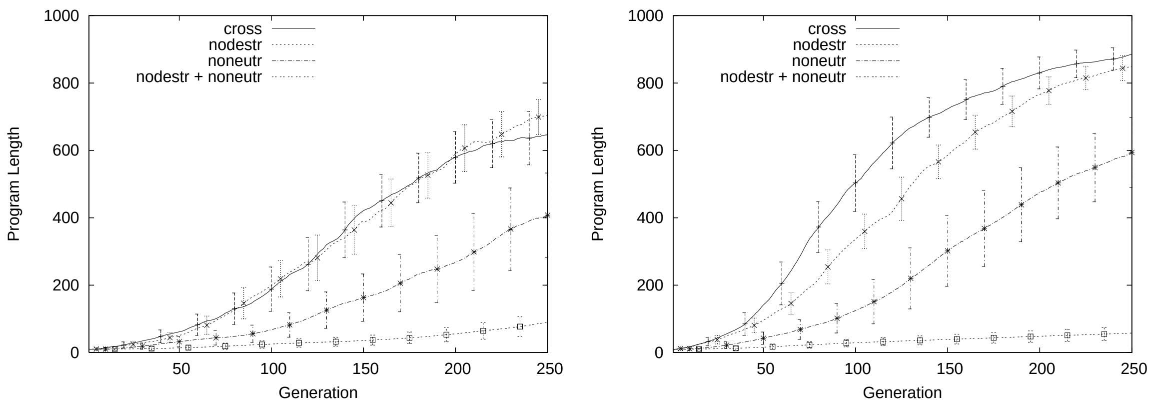

Figure 9.2: Development of average absolute program length when using crossover (cross) almost without a restriction by the maximum program length (1000 instructions). Code growth more reduced without neutral variation effects than without destructive effects. Bars show standard deviation of program length in the population. Average figures over 30 runs for *mexican hat* (left) and *spiral* (right).

Semantic Diversity

We have seen above that the *average fitness of best solutions* changes only little if destructions are not accepted. This is quite different for the *average fitness in the population* as a comparison between Figures 9.3 and 9.4 reveals. By including the destructive crossover results the average fitness develops much more diverse and much more different from the best fitness. Note that the standard deviation applies to the fitness values in the popula-

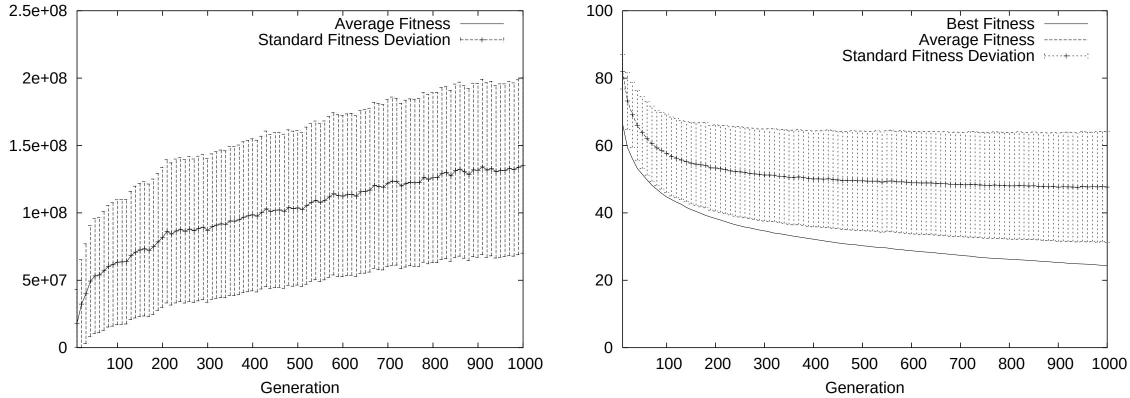

Figure 9.3: Development of average fitness and standard deviation in the population for *mexican hat* (left) and the *spiral* (right) using crossover (cross). Standard deviation is printed 5 times smaller for *mexican hat*. Average figures over 100 runs.

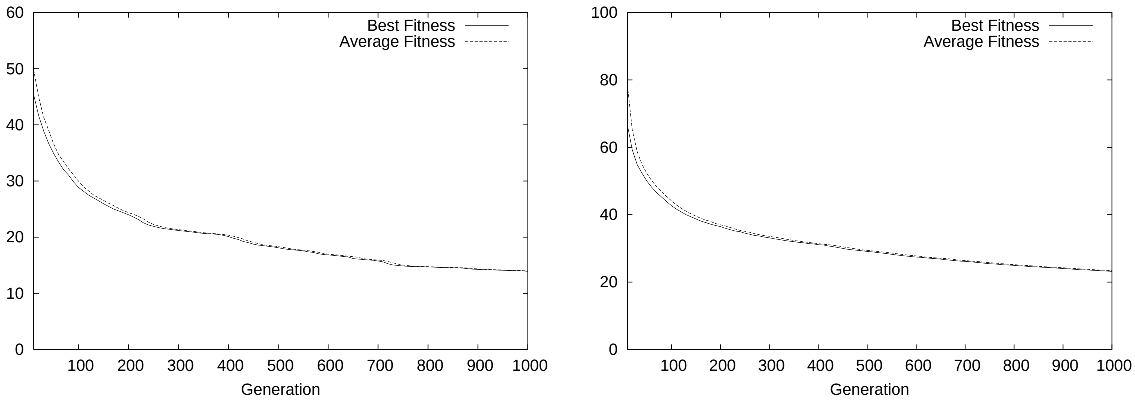

Figure 9.4: Development of average fitness and best fitness for *mexican hat* (left) and *spiral* (right). very similar if destructive variations are canceled (nodestr). Standard deviation is below 1 (not printed). Average figures over 100 runs.

tion, not to the development of average fitness over multiple runs. Typically, the difference between average fitness and best fitness is more significant for the continuous problem in contrast to the discrete task with its more narrow range of fitness values. The development of average fitness in noneutr runs, by contrast, has not been found very different from the development in normal runs (not documented).

For both problems average fitness and best fitness are almost congruent in Figure 9.4 if worse offsprings are excluded from the population (nodestr). Then most individuals in the population share the same fitness value. A low standard deviation of fitness values is an indication for a low semantic diversity of programs in the population. Accordingly, the diversity of the effective code (structural diversity) in the population may be expected lower, too. This is due to much less effective variations of individuals that reach the population and because most neutral variations alter the noneffective code only. Even if better individuals are selected more frequently (because more exist in the population) the low diversity may reduce the probability for improvements. If a better individual occurs it will most likely become the best individual of the population, too, and the population follows this new fitness gradient quickly. Additionally, (effective) code may spread faster in the population because worse offsprings cannot overwrite better tournament losers.

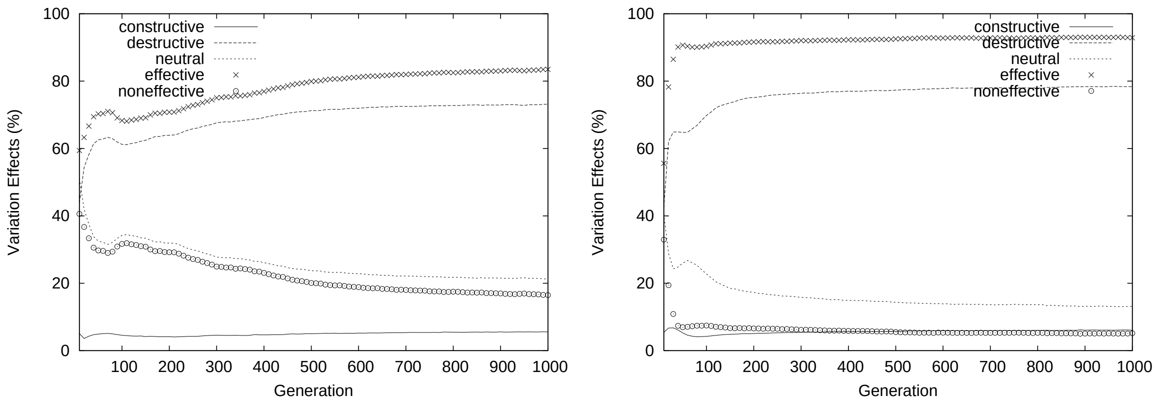

Figure 9.5: Development of crossover effects (cross) for *mexican hat* (left) and *spiral* (right). Average figures over 100 runs.

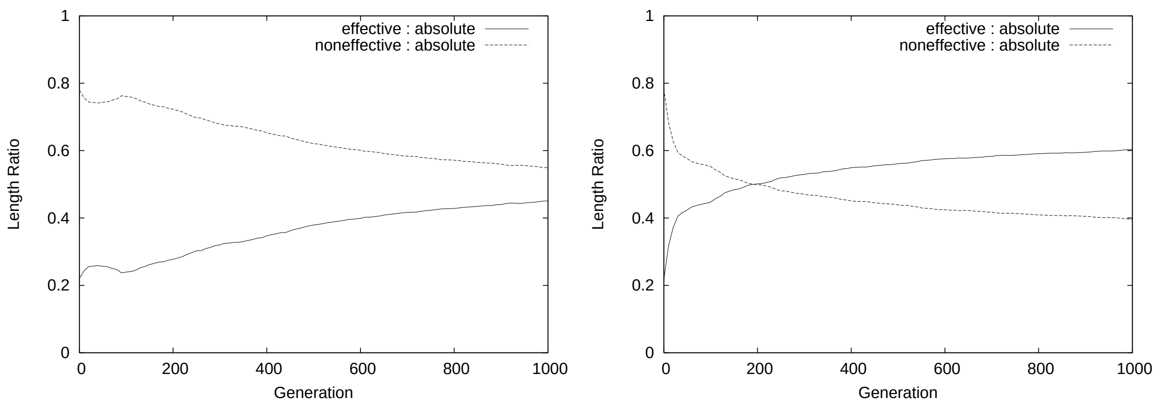

Figure 9.6: Development of length ratios with crossover (cross) for *mexican hat* (left) and *spiral* (right). Average figures over 100 runs.

9.8.4 Development of Variation Effects

It has already been demonstrated in Sections 5.9.4 and 5.11.4 that larger variation step sizes may lead to a higher proportion of noneffective code in programs. Especially when using multiple instruction mutations this does not necessarily produce larger programs, too, which is a clear experimental evidence of the protection effect in terms of a reduction of effective step size. The protection effect has also been held responsible for promoting the creation of semantic introns with crossover in Section 5.9.1 after all structural introns have been removed. Moreover, a better protection was achieved in terms of smaller effective step sizes by increasing the proportion of explicit introns (see Section 5.9.5).

In the following we are interested in how the proportions of structural and semantic variation effects (see Section 5.1), on the one hand, and the proportion of (non)effective code, on the other hand, develop over a run. This is demonstrated for a segment variation operator (cross) and for instruction mutations (mut). How does the protection effect influence the growth of code and the development of introns ? Is the protection effect reinforced in the course of a run (by a higher rate of neutral variations and/or neutral code) ?

In genetic programming typically a high amount of crossover operations result in offsprings which fitness is worse than the fitness of their parents. On average, about 70 to 90 percent

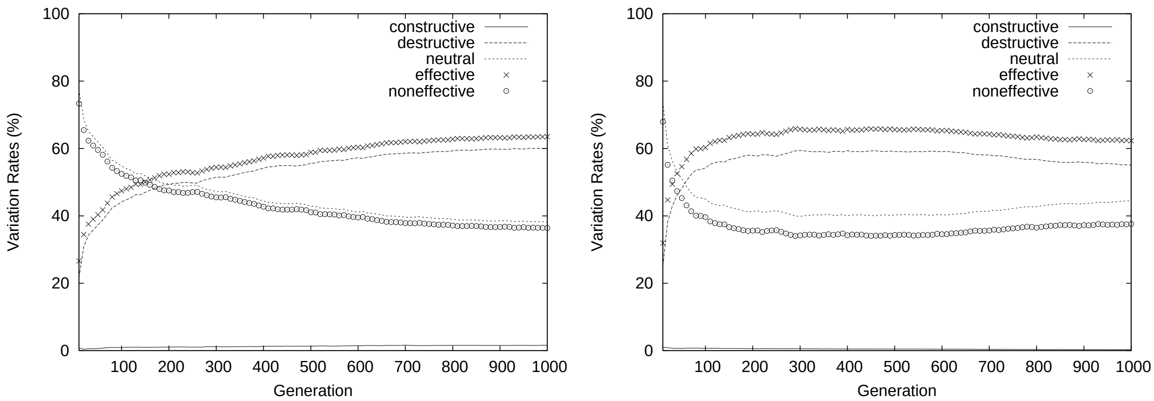

Figure 9.7: Development of variation effects with instruction mutations (mut, B0) for *mexican hat* (left) and *spiral* (right). Average figures over 100 runs.

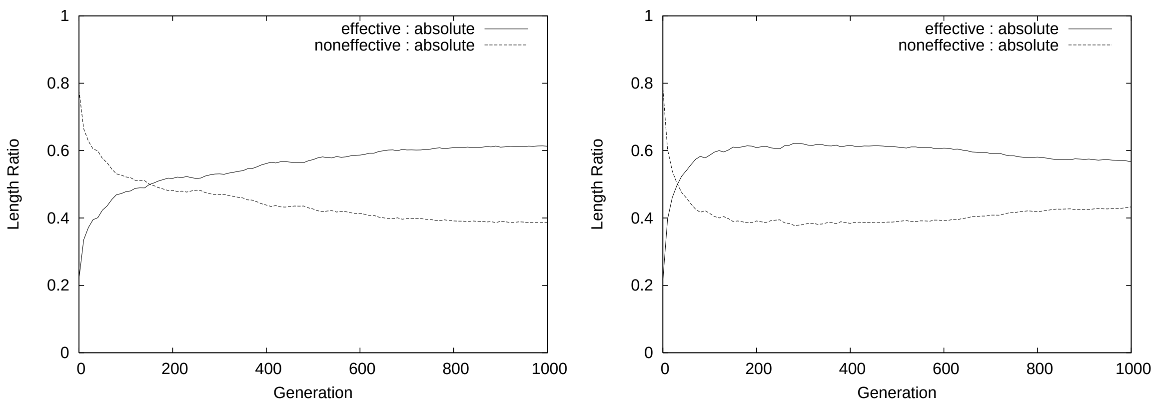

Figure 9.8: Development of length ratios with instruction mutations (mut, B0) for *mexican hat* (left) and *spiral* (right). Average figures over 100 runs.

of all crossover variations of a run are destructive when using unrestricted linear crossover (see Section 5.9.3). Nordin and Banzhaf [65] argue that the ratio of effective length and absolute length of a program $\frac{l_{eff}}{l_{abs}}$ is related to the probability that (unrestricted) linear crossover will be destructive. An increasing proportion of noneffective code is supposed to increase the rate of neutral crossover, i.e., two segments are exchanged which act as intron code in both the parents and the children. We use the information about the structural intron code to verify this correlation. While such a correlation is not doubted here in principle, a relevant increase of neutral variations has not been observed.

The two counterexamples in Figure 9.5 reveal that the destruction rate does not drop over a period of 1000 generations when using linear crossover as defined in Section 5.7.1. On the contrary, destructive operations mostly increase until their number converges to a certain maximum. The rate of neutral operations decreases, accordingly, while the rate of constructive operations is constantly low. Neutral and destructive variation effects are correlated with the rates of effective variations and noneffective variations, respectively. These structural variation effects are in turn correlated with the proportions of effective code and noneffective code in programs (see Figure 9.6). Both correlations are stronger for the *mexican hat* problem due to less less effective neutral variations and, therefore, less semantic introns.

Only during the first generations the proportion of effective code in Figure 9.6 as well as the proportions of effective and destructive variations in Figure 9.5 decrease. This is true until about generation 100 for the *mexican hat* problem. Only up to that point the growth of programs is unrestricted by the maximum size limit (compare Figures 5.5 and 5.6). After most programs have reached the maximum size, both the rate of destructive variations and the extent of destruction on code level increase again. These effects are weaker with the *spiral* problem because of a more rapid (and stronger) increase of effective code already at the beginning of a run. Semantic introns may emerge almost as easily as structural introns here which results from both the discrete fitness function and the use of branches.

We may conclude that the protection effect increases the rate of noneffective code (if ever) only at the beginning of a run until the program lengths are maximum. Note that an additional drift of intron code may not be excluded during this initial grow phase. This is especially true because the variation step size is unrestricted and the rate of neutral variations is highest in the early generations. In the following generations the intron rate decreases since the fitness force lets the effective program length grow further. Apparently, this force is stronger than the protection force on the intron rate. As demonstrated in Chapter 6, even with a larger maximum size limit or more registers, i.e., longer or wider program graphs, the proportion of effective code does not decrease over a run.

We have seen that the ratio of noneffective code influences the ratio of neutral variations if the segment length is unrestricted. But both ratios develope still quite differently. We will now demonstrate that this is different for instruction mutations, i.e., minimum absolute step sizes. Figures 9.7 and 9.8 show that, in this case, the average proportions of neutral variations and introns are almost identical since introns are almost only created by neutral variations. Both proportions decrease although code growth is not limited by the maximum program bound here. (The average length in generation 1000 is about 70 only.) The decrease is much faster at the beginning than towards the end of a run. For *spiral* there is even a slight increase at the end after a longer period of stagnation.

9.8.5 Neutral Drift ?

Figures 9.9 and 9.10 show two characteristic example runs with the *spiral* problem and instruction mutations (mut). The development of the best fitness reflects approximately the progress in the population. Longer stagnation phases of the best fitness, as those occur especially with discrete problems, are correlated with periods of many neutral variations. Actually, the rate of neutral variations increases continuously here during such exploration phases while the rate of destructions decreases, accordingly. As a result, the noneffective neutral code grows in the population individuals. One can see that both neutral code and neutral variations react only slightly delayed for a few generations to a new (best) fitness situation.

If a better (effective) solution occurs this may spread rapidly within a few generation. That is, the population follows (exploits) a newly detected positive fitness gradient. Interestingly, the amount of noneffective code drops again together with the number of neutral variations in this case. Almost simultaneously, the effective length increases which is reflected by a stepwise progression in Figure 9.10. Such an observation may be explained by reactivations only. After a period of neutral (and destructive) variations the "silently" developed neutral code is suddenly reactivated in a constructive way. During such *neutral walks* over plateaus of the fitness landscape the individual structure may be developed continuously (in quality and size) by neutral changes while destructive offsprings more likely extinct, i.e., be replaced within the population.

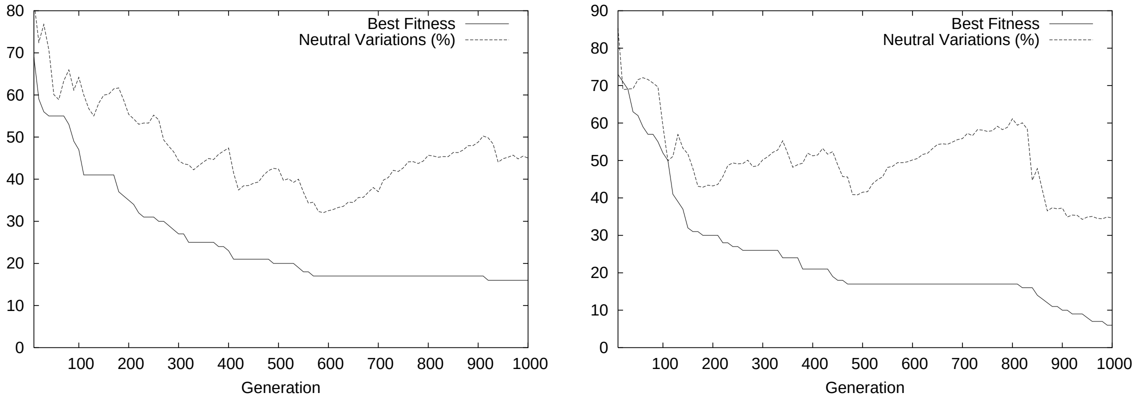

Figure 9.9: *spiral*: Development of best fitness and the rate of neutral variation over two typical example runs using instruction mutations (mut, B0). Rate of neutral variations increases almost only on fitness plateaus (during stagnation periods of the best fitness).

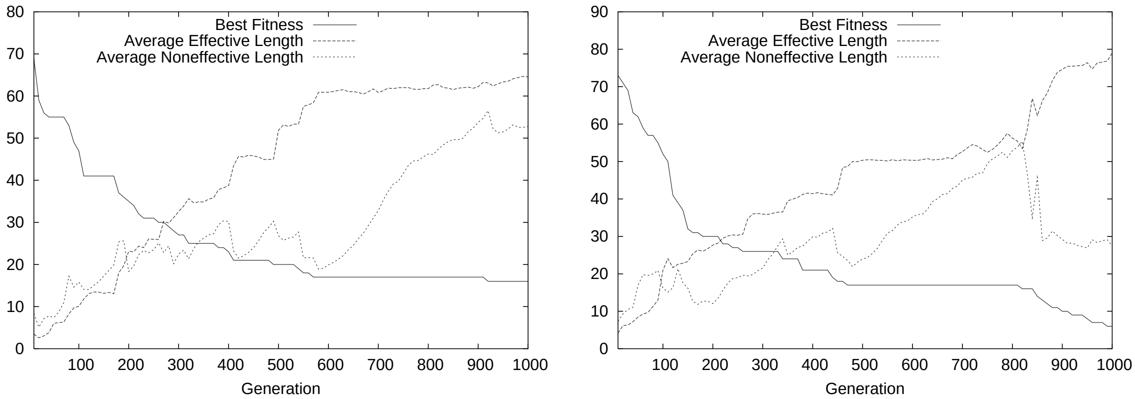

Figure 9.10: *spiral*: Development of best fitness, average effective length, and average noneffective length over the same runs as in Figure 9.9. Neutral noneffective code grows continuously on fitness plateaus and shrinks on fitness gradients. Effective code grows stepwise. Length figures have been slightly shifted in vertical direction for a better view.

The fact that reactivations of intron segments improve the (best) fitness shows that introns do not only contribute to unnecessary code growth, but are relevant for evolutionary progress and for the growth of effective code, too, and so are neutral variations. In particular, the results demonstrate that the structural noneffective code (created by noneffective neutral variations) is used for solution finding, at least with random instruction mutations. Similar correlations as in Figures 9.9 and 9.10 may be supposed for the development of *effective* neutral variations and semantic introns.

Since instruction mutations reduce the step size of macro variations to a minimum, neutral variations are a necessary condition for code growth *and* for the evolutionary progress. Moreover, intron code emerges almost exclusively from neutral variations in this case. The above analysis of single runs has shown how neutral variations, code growth and fitness progress are connected. But what is the driving force that lets both neutral variations and neutral code increase during phases where the best fitness stagnates ? Two possible theories may be valid here.

(1) Neutral variations preserve the semantics of a solution and, therefore, guarantee a high survival rate of offsprings. Actually, since the survival rate of offsprings has been found

very low after destructive variations and since the rate of constructive variations is low, too, mostly individuals will be selected that result from neutral variations.

If the best fitness stagnates the population explores a plateau region of the fitness landscape more widely while the proportion of neutral variations increases. Another important reason why neutral variations have a high impact on the growth of intron code is that (the size of) this code does not influence the program fitness directly. Especially the structurally noneffective code emerges relatively easy in linear GP. Thus, introns may be argued to grow by a random drift if the population spreads over a plateau of the fitness landscape.

As mentioned in Section 9.4, Kimura's [48] neutral theory considers a random genetic drift of neutral mutations as a main force of natural evolution. Accordingly, a *neutral drift theory* of code growth may regard a drift of intron code by neutral variations as a dominating force of code growth. At least this may play an important role for instruction mutations.

(2) By applying only deletions or insertions of single instructions a possible influence of a protection effect in terms of a reduction of effective step size is restricted as far as possible, as discussed in Section 9.3. However, protection may still occur here such that a high proportion of neutral code increases the probability for neutral variations. This effect may also be responsible for the growth of intron code on fitness plateaus since it lets (effective) programs with a higher intron rate survive for a higher probability.

9.8.6 Crossover Step Size

For the following considerations the reader may recall that linear genetic programs as used in this thesis may be represented as an equivalent directed acyclic graph (DAG, see Section 3.3). The maximum width of such a graph is restricted by the available number of registers while the maximum depth is limited by the number of instructions, i.e., inner graph nodes. In narrow graphs more program paths are affected, on average, by linear crossover when exchanging instruction segments on the imperative level. Then most segments may separate the "linear" graph structure almost completely.

This let us assume that the influence of a segment on the fitness depends only partly on its length. At least, linear crossover might not be significantly more destructive beyond a certain segment length. The *relative fitness change* is defined as the difference in fitness between parent and offspring (*absolute fitness change*) divided by the parental fitness:

$$\frac{\mathcal{F}_p - \mathcal{F}_o}{\mathcal{F}_p} \tag{9.1}$$

The average fitness change is usually negative since much more variation effects are destructive than constructive. Recall that the optimum fitness value $\mathcal{F}$ is zero.

Figure 9.11 confirms our assumption. In a linear genetic program the segment length (structural step size) is proportional to the fitness change (semantic step size) only to a certain degree. Even if only the relative fitness change is printed in Figure 9.11 this has been found to be true for absolute fitness changes as well. One can see that the more registers are provided the larger is the segment length beyond that the average fitness stagnates.

To achieve that the average segment length stays the same over the whole run, the program length is constant here. Thus, crossover exchanges equally-long segments between two individuals right from the beginning. Nevertheless, crossover steps become relatively more destructive over a run. Figure 9.12 compares the development of relative fitness changes.

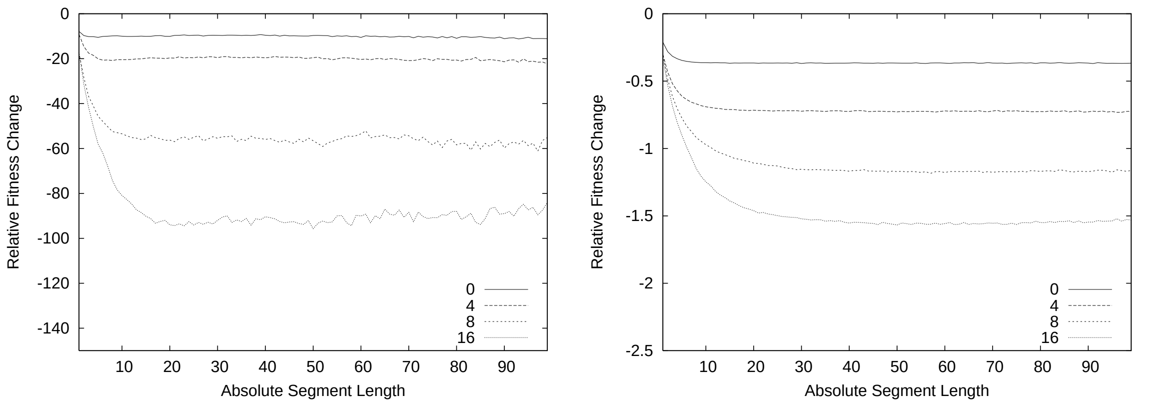

Figure 9.11: Average relative fitness change per segment length when using crossover (cross) and a constant program length of 200 instructions. Larger segments do not become more destructive beyond a certain segment lengths depending on the number of calculation registers (0, 4, 8, and 16). Average figures over 30 runs for *mexican hat* (left) and *spiral* (right).

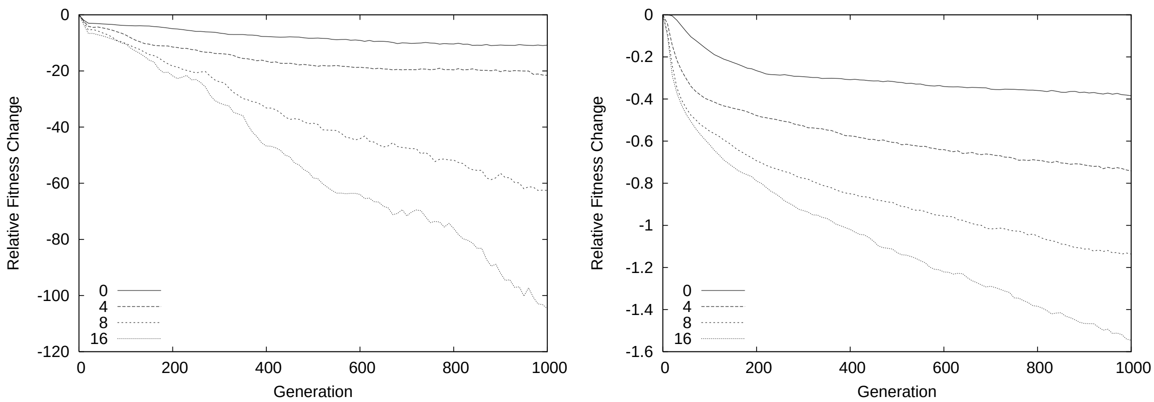

Figure 9.12: Development of the average relative fitness change when using crossover (cross) and a constant program length of 200 instructions. Crossover becomes relatively more destructive during run. Average figures over 30 runs for *mexican hat* (left) and *spiral* (right).

The more the program fitness improves the larger is the relative destruction. This is all the more valid the more registers are used for calculations. Using the absolute fitness change is less appropriate here because it necessarily decreases in the course of a run. Also note that very similar figures may be produced for two-segment mutations.

9.8.7 Implicit Bias: Linear Crossover

Let a variation operator be free from an *explicit bias* if there is no significant code growth without fitness. That is, on average, the same amount of code is added as it is removed from the programs. The exchange of subprograms between individuals during crossover may not increase the average program size in the population. In contrast to crossover, subprogram mutations have to be implemented explicitly such that the average program size in population is not changed. This has been realized in Section 5.7.5 by selecting the segment length in relation to the length of another randomly selected individual. We refer to an *implicit bias* here if program growth is forced in the presence of fitness only, but does not result from an explicit configuration of variation parameters.

As noted in Section 9.2.3, a removal bias has been argued to be a direct cause of code growth in tree-based genetic programming when using subtree crossover. Such an implicit grow bias results from the fact that the removed subtree may cause a fitness change that depends on the subtree size *in relation to* the program size (relative subtree size). The fitness change caused by the added subtree, instead, is more independent from its size. One reason for this is the single connection point (edge) at which all subtrees may influence the result of the main program.

The situation is less clear when using crossover in linear GP. There are several reasons why the effect of an inserted instruction segment is *not* independent from its length. First, the more instructions are removed from or inserted in a linear program the more (effective) register contents may be changed, on average. Remember that register manipulations correspond to modification of edges in the graph representation of a linear genetic program. Thus, the longer an inserted instruction sequence is the more variation points may be affected on the functional level.

Second, the available number of registers determines the maximum width of the (effective) DAG (see Section 6.1). The wider the program graphs are the less program paths (variation points) may be modified At least theoretically a removal bias becomes more likely then. Since linear crossover works on instruction level, however, it is rather unlikely in general – especially with many registers – that exchanged instruction segments form contiguous subgraphs.

Third, not all register manipulations will be effective, since usually not all instructions of an inserted or deleted segment contribute to the effective code. It may be demonstrated easily that the *effective* length of crossover segments is approximately the same for insertions and deletions. The average effective segment length strongly depends on the total rate of effective instructions in program. This is true for both types of operation, segment deletions and insertions. In general, it depends on the program context, how many segment instructions will be effective. For insertions, this is influenced by the number of effective registers at the insertion point and the number of registers manipulated by the segment code as a whole. Additionally, it is important how much the segment instructions are interconnected on the level of register dependences.

Fourth, The directed graph structure allows inserted components not only to be used by the program but to use parts of the program graph itself. This happens the more likely in linear programs the less registers are available, i.e., the more the graph is restricted in width (see Section 3.3). In more narrow ("linear") graphs more paths lead from the root through an instruction node than in wider graphs (or trees). At least in the former case, we may not expect that an insertion is less destructive than a deletion of equal size.

Recently, Soule and Heckendorn [90] gave an experimental evidence of the removal bias theory in tree-based GP. We repeat the experiment here for the linear program representation and linear crossover. Basically, the correlation between the relative fitness change and the relative segment length is calculated separately for inserted and deleted segments. The *relative segment length* denotes the absolute length of an inserted (deleted) segment as a percentage of the length of the destination (source) program. Note that an inserted segment may exceed the size of the destination program. However, since this situation does not occur very often it may be neglected here.

A removal bias may only be relevant for linear crossover or two-segment variations in general if the lengths of the inserted segment and the deleted segment may be different. Due to the influence of the maximum length bound, however, this period will not last very

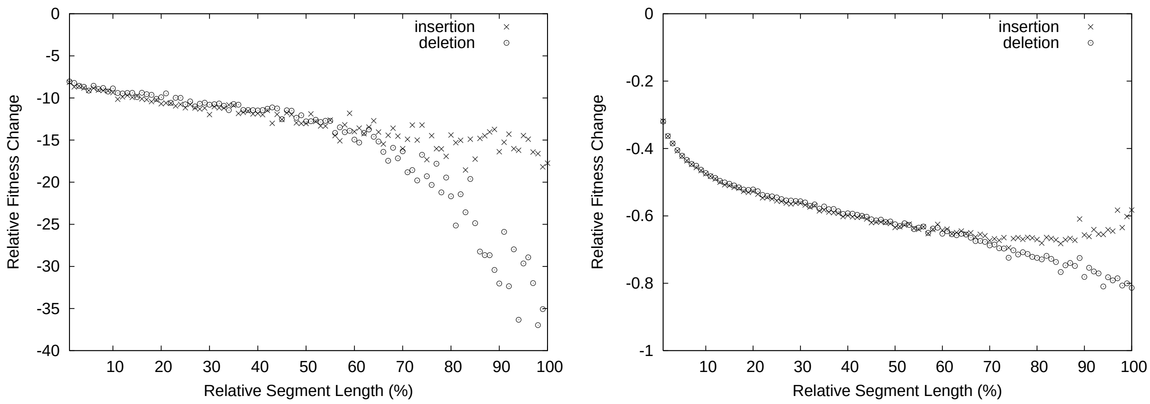

Figure 9.13: Average relative change in fitness per relative length of the inserted and the deleted crossover segments (cross). Average figures over 30 runs for *mexican hat* (left) and *spiral* (right).

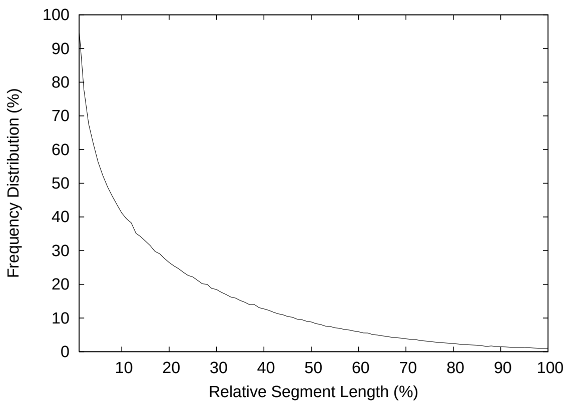

Figure 9.14: Frequency distribution of relative and absolute lengths of inserted and deleted crossover segments (cross). Average figures over 30 runs for *mexican hat* (similar for *spiral*).

long. Recall that in our crossover implementation equally-long segments are exchanged if an offspring would exceed the maximum size limit, otherwise. Therefore, we allow a maximum program length (as well as a maximum segment length) of 1000 instructions in this section. This guarantees that programs may grow almost unrestrictedly. At least, the average length does not reach the maximum within the 250 generations observed here (see Figure 9.2).

In Figure 9.13 a removal bias occurs only for relative segment lengths larger than 70%. For two reasons it may be questioned that such a bias has a relevant influence on code growth when using linear crossover. First, programs resulting from larger destructions may be selected only for a low probability, as noted before. Thus, large destructive variations may hardly contribute to code growth or be relevant for the evolutionary progress. Second, such large relative segment lengths do not occur very frequently as we learn from the frequency distribution in Figure 9.14. Only the distribution of absolute segment lengths depends on the absolute program length such that the probability for selecting shorter segments decreases with the program length.

9.8.8 Implicit Bias: Effective Instruction Mutations

In Section 5.11.3 we have seen how an explicit grow bias influences both code growth and prediction performance if instruction mutations are applied. Now we will investigate whether such mutations are implicitly biased even if instructions are deleted or inserted for the same probability. Is deletion of a single instruction more likely destructive than insertion ? If a randomly selected instruction is deleted, it depends on the proportion of (non-)effective instructions in a program whether the deletion is effective or not. If a random instruction is inserted at a program position, its destination register will be effective depending on the proportion of registers that are effective at that position. In a larger intron block the average number of effective registers is rather low. Thus, if an instruction is inserted in a context of other introns the probability that a new instruction becomes an intron may be expected higher. Such interactions lead to similar proportions of semantic and structural variation effects for instruction deletions and instruction insertions (not documented).

Let us now consider (explicitly) effective instruction mutations as described in Section 5.10.4. Recall that the deletion of an effective instruction node comprises the removal of several edges from the corresponding program graph – one for each operand register and at least one for the destination register – while each removed edge (register) may lead to disconnections (deactivations) of code. During an effective insertion, instead, only the choice of the destination register can be a source of deactivation. This happens if another instruction becomes inactive that uses the same destination register. The operand registers, instead, just add new register dependences to other instruction, i.e., edges to the effective graph component. This may result in reactivations of formerly inactive code but not in deactivations (see also Section 5.10.5). Since the rate of inactive instructions is usually low with effective mutations reactivations may occur less frequently than deactivations.

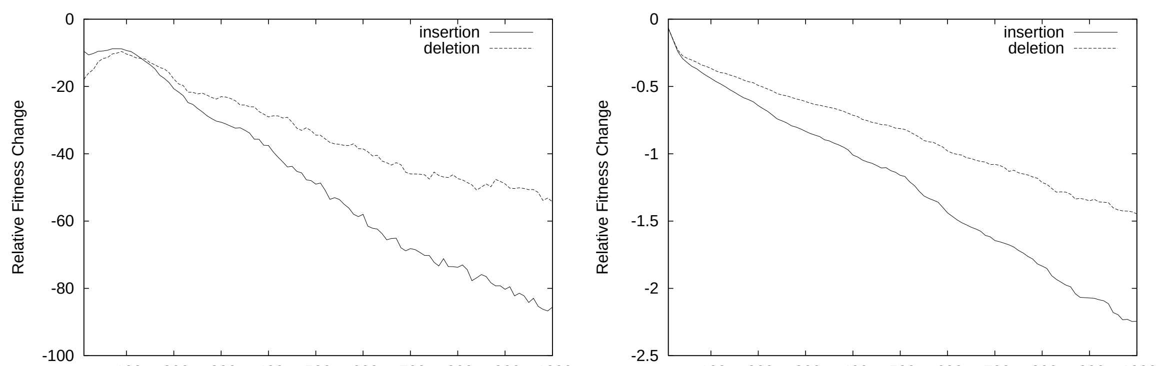

Figure 9.15: Development of the average relative fitness change for *mexican hat* (left) and *spiral*) (right) when using effective instruction mutations (effmut). Insertions more destructive than deletions (implicit shrink bias). Average figures over 30 runs.

Interestingly, experimental results show that effective insertions lead to *larger* semantic variation step sizes, i.e., a larger average fitness change, than effective deletions (see Figure 9.15). As indicated before, the proportion of destructive variations is approximately the same for both variations. Apparently, effective deletions are less destructive because the effective code stabilizes over a run (as demonstrated in Chapter 8). Consequently, this imbalance leads to an *implicit shrink bias* or *insertion bias*.

An implicit shrink bias may be another reason why the absolute size of programs stays small if only effective code is created – besides the fact that noneffective instructions are not inserted directly. In principle, a shrink bias occurs with random instruction mutations, too. However, noneffective variations create much more noneffective code with these operations. Note that, by our definition, an implicit bias may affect non-neutral variations only, i.e., variations that change the fitness. This means for one-instruction mutations that an implicit bias influences the growth of effective code rather than the number of introns (see also Section 9.3).

9.9 Control of Code Growth

We discuss different possibilities how code growth may be controlled implicitly or explicitly in linear genetic programming. Basically, we distinguish between a control of code growth by variation or selection. The following section summarizes results from Chapter 5 and this chapter concerning the influence of different variation operators and variation parameters on code growth. Additionally, the phenomenon is analysed why code growth occurs to be so much more aggressive with segment recombination than with segment mutation.

9.9.1 Variation-Based Control

As defined in Section 5.3 the absolute variation step size denotes the amount of code that is deleted and/or inserted during one variation step. Because a deletion and an insertion are always applied together during a crossover operation (cross) or a two-segment mutation (segmut), the possible speed of code growth depends on the maximum *difference* in size between the deleted and the inserted segment (see Section 5.9.4). Obviously, there is no code growth possible if this difference is set to zero. Another possibility to limit the length distance between parent and offspring is to use a smaller maximum segment length which indirectly restricts the maximum difference of segment lengths. Linear crossover may not be explicitly biased towards creating larger or smaller programs since it only moves existing code within the population by a mutual exchange between individuals. Hence, the average program length may not be changed by crossover. Segment mutations must be explicitly configured such that newly created segments are not larger than deleted segments, on average.

A control of code growth by removing (structural) introns explicitly from the population individuals (effcross) turned out to be insufficient for linear crossover. Mostly the protection effect leads to an increase of other (semantic) introns in programs then. Depending on the configuration of the instruction set this replacement may let programs become similarly large. Besides, the processing time is increased since, in general, semantic introns may not be detected efficiently and removed before the fitness evaluation during runtime.

The more probabilistic one-segment recombination operator (oneseg) as well as one-segment mutations (onesegmut) either insert *or* delete a segment for certain independent probabilities. Unlike two-segment variations there is no substitution of code. This allows the speed of code growth to be controlled by an explicit bias. For instance, a shrink bias may be induced either by allowing bigger parts of code to be deleted, on average, or (better) by applying deletions of code more frequently than insertions. The latter variant does not increase the average variation step size in contrast to the former one.

Figure 9.16 compares code bloat for one-segment variations (almost) without a maximum limitation of program length. More precisely, the maximum limit of 1000 instructions influences code growth only slightly over a period of 250 generations. In general, no

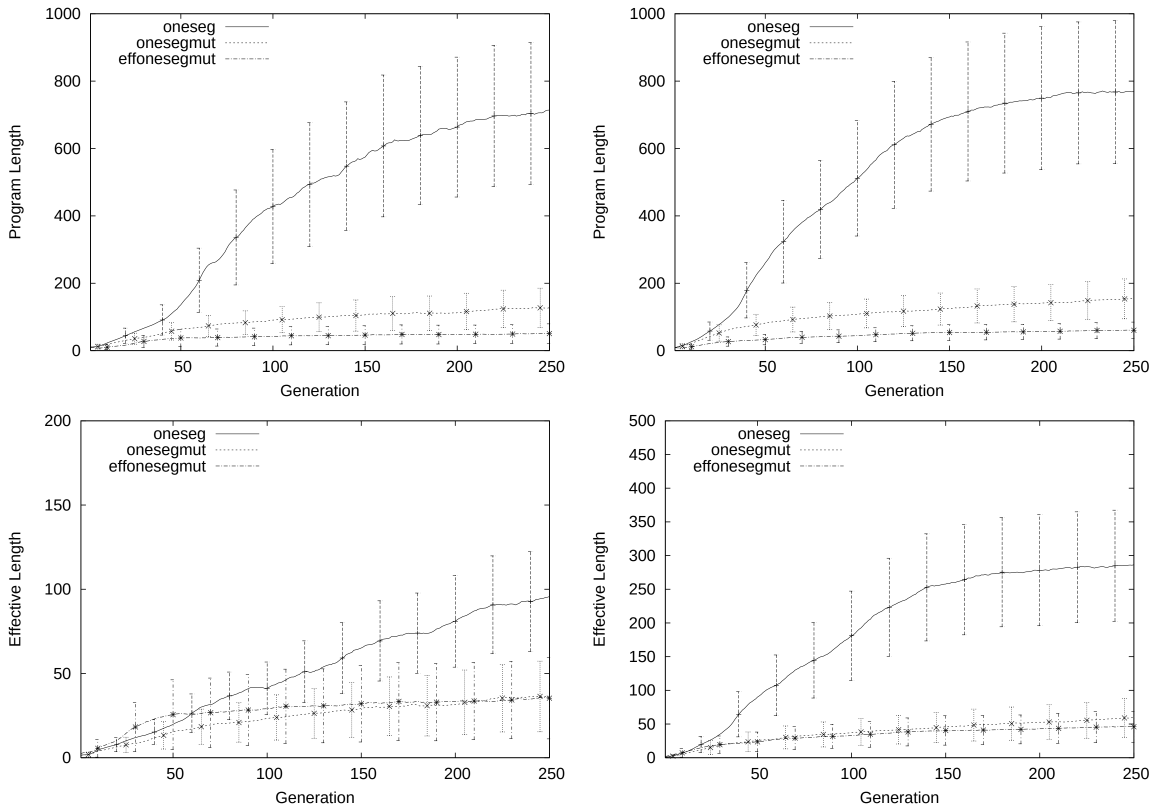

Figure 9.16: Development of average (effective) program length when using one-segment variations (oneseg, onesegmut, effonesegmut) with a maximum program length of 1000 instructions. Programs significantly smaller with randomly created segments. Bars show standard deviation of program length within the population. Average figures over 30 runs for *mexican hat* (left) and *spiral* (right).

influence may be expected until the program lengths exceed $\frac{l_{max}}{2}$ with l_{max} is the maximum program length. Until that point, selected segment lengths are smaller than the remaining program space. It is an important result, that recombination leads to a much faster and larger code bloat here than mutations even if for both variation types the segment length and, thus, the absolute step size is limited only by the program length. Reasons for this will be discussed below. Hence, using mutation instead of recombination forms one out of three methods reviewed here to limit the influence of a protection effect or a drift effect on the growth of (intron) code. Note that the relative difference in effective code may be smaller (but still significant) since this code depends more strongly on the problem fitness. For the discrete *spiral* problem the effective code grows larger also because the applied function set allows semantic introns to be created much more easily.

Also note that similar observations have been made when comparing code growth of two-segment recombination (crossover) and two-segment mutation (not shown). The difference in program size between recombination and segment mutations occurs to be smaller in Sections 5.9.1 and 5.9.2 due to a lower maximum bound only (200 instructions). This has been used to assure a comparison of prediction errors that is not too much depending on the program size.

By reducing the absolute mutation step size to one instruction (mut) a less explosive increase of program length is possible as this occurs if a large number of instructions is

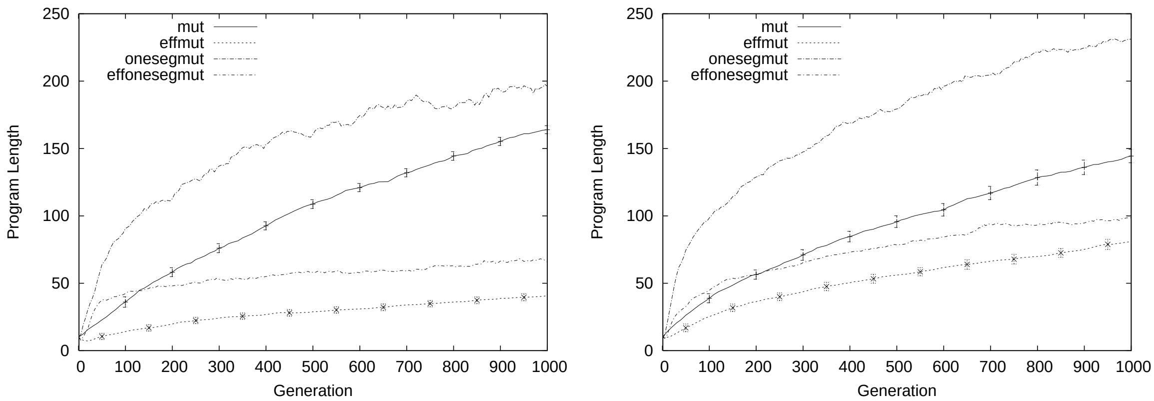

Figure 9.17: Development of average program length when using instruction mutations (mut, effmut) compared to segment mutations (onesegmut, effonesegmut) without a maximum limitation of program length. Programs significantly smaller if only effective instructions are created. Small difference in program length between using minimum or maximum segment lengths, especially with effective mutations. Bars show standard deviation of program length within the population. Average figures over 30 runs for *mexican hat* (left) and *spiral* (right). Configuration: 100% macro mutations without explicit length bias (B0).

allowed to be deleted or inserted per variation step. One reason for this is that evolution may not further reduce the destructive influence (effective step size) of deletions implicitly by producing more intron code. In this way, the evolutionary advantage of both structural introns and semantic introns is suppressed. Instead, the probability for neutral variations is increased by both smaller step sizes and more intron code. In general, a smaller absolute step size acts as a second measure against code growth.

It is interesting to see that the difference in average program size between unrestricted one-segment mutations (maximum step size) and one-instruction mutations (minimum step size) is smaller than this might have been expected (see Figure 9.17). This may be taken as another hint that the variation step size influences code growth only indirectly (see also Section 9.3). An influence by the maximum size bound (1000 instructions) can be excluded here for all mutation operators, simply because programs remain significantly smaller. Moreover, none of the operators is explicitly biased towards creating larger solutions already on its own, i.e., insertions and deletions are applied for 50 percent each. When applying recombination with a minimum segment length of one instruction programs grow similarly as with instruction mutations (not shown). Thus, the relative difference in program growth is much smaller compared to using segments of arbitrary length for both variation types.

A *direct* insertion (and variation) of noneffective instructions is avoided by inducing effective mutations exclusively (effmutX). That is, newly created instructions are always effective. Then noneffective code (structural introns) may only result from *indirect* deactivations of depending instructions. The avoidance of noneffective neutral variations leads to a significant reduction of noneffective code in particular. In this way, the effective mutation operator realizes an *implicit control of code growth* in linear GP. Actually, it makes the size of program solutions depend more on their fitness than on variation and be closer to the required minimum size.

Alternatively code growth is reduced, if only the direct creation of structural introns is disabled while the mutation step size is unrestricted. This is done by (fully) effective segment mutations (effonesegmut). In Figure 9.17 the absolute program length develops not even half as large as if segments are created completely by random (onesegmut). The effective lengths are rather similar, however (not shown). Avoiding a direct insertion of (structural) intron code denotes a third possibility to reduce code growth. This is true even though semantic intron formation could still increase the complexity of programs by acting as a protection of the (semantically) effective code. One explanation may be that the creation of large semantic introns is more difficult than the creation of structural introns. Another possible explanation is that structurally noneffective instructions may be deleted but not directly inserted here during a genetic operation which corresponds to an explicit shrink bias in terms of this type of intron code.

9.9.2 Why Mutations Cause Less Bloat

An interesting question that arises when analysing code growth in linear GP is why so much smaller programs occur with (segment) mutation than with recombination although the segment length is not explicitly restricted in both cases. Instead, the *proportion* of (non)effective code in programs (and segments) is similar over a run for both kinds of variations
In the following paragraphs we summarize different hypotheses which may explain this phenomenon and support them by experimental results. In general, causes given here represent preconditions for code growth rather than driving forces (see Section 9.2). Nevertheless, these conditions may significantly increase the influence of a driving force on the size of solutions.

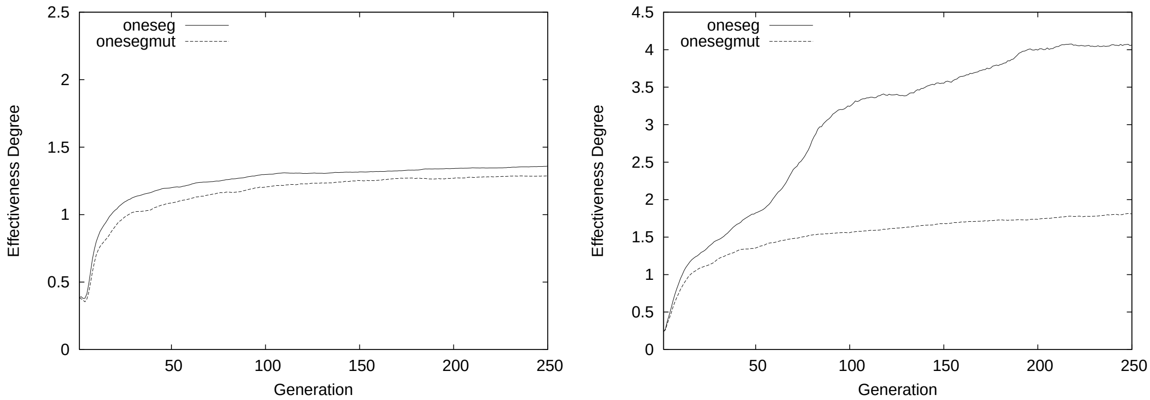

Figure 9.18: Development of effectiveness degree over the segment length when using recombination (oneseg) or mutation (onesegmut). Higher effectiveness with recombination and *spiral*. Average figures over 30 runs for *mexican hat* (left) and *spiral* (right).

(1) One explanation for a stronger code growth by recombination might be that it uses only material from the population. This facilitates a stabilization of the (functional) program structure over a run in contrast to insertions of large random segments. We have seen in Section 8.7.2 that the effectiveness degree, i.e., the dependence degree of effective instructions, increases over a run. This may be expected at least in part for the noneffective instructions, too. Such introns may form less and larger graph components with a higher (in)degree of nodes. Instead, if large random segments are inserted, program structures might become less robust because the dependence degree of (effective and noneffective) instructions is lower in general. As a result, depending program instructions are more likely deactivated or reactivated, respectively, during variations by what the effective step size

may be increased. In particular, larger (effective) programs may produce offsprings with a lower fitness (see cause (2)). The reader may recall that the situation may be different for restricted (or minimum) mutation step sizes which are very well able to create robust program structures with a high dependence of instructions (see Section 8.7.2).

But not only the *implementation* of large robust (intron) code may be restricted by using segment mutations. Obviously, also a *propagation* of code in the population is not possible – at least not by the variation operator – if segments are created randomly.

The above assumptions are partly confirmed by the results in Figure 9.18. At least for the *spiral* problem the dependence degree of effective instructions is significantly higher with (one-segment) recombination than with mutation. Recall that instruction dependence is usually higher if programs include branches (see Section 3.4). We have not calculated the dependence degree of noneffective instructions that may be more different in case of the *mexican hat* problem.

On a structural level subtree mutation and recombination are more similarly destructive in tree-based GP, since the indegree of tree nodes is constantly 1, by definition. Correspondingly, the effect of both operators on code bloat may be more similar than this is found in linear GP.

(2) The average fitness of individuals in the population should be higher than the fitness of equally sized random programs. We may assume that this is true for arbitrary large subprograms (building blocks), too. Thus, a lower fitness change (semantic step size) may be caused by segments that originate from another population individual than by segments that are created randomly.

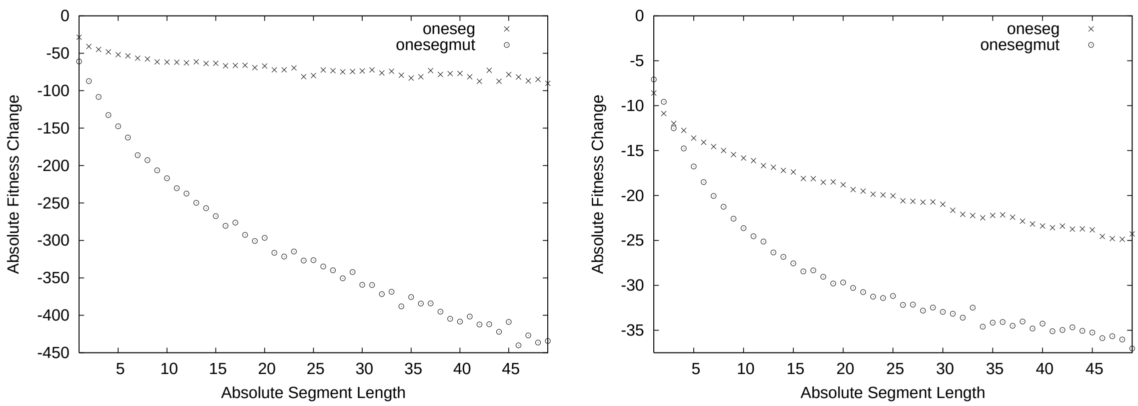

Figure 9.19: Development of fitness change over the segment length when using recombination (oneseg) or mutation (onesegmut). Mutation increasingly more destructive than recombination for larger segment lengths. Average figures over 30 runs for *mexican hat* (left) and *spiral* (right).

Figure 9.19 compares the average fitness change between recombination and mutation. Especially for *mexican hat* mutated segments turn out to be much more destructive than recombined segments of equal size. This difference increases with the segment length. Since the diversity of population code is usually lower than random code, more similar segments may be exchanged (only) by recombination. The number of identically exchanged instructions between individuals increases with the segment length already because more identical program positions may be affected. Interestingly, even if recombined segments cause smaller semantic step sizes than random segments, their structural step sizes are larger, on average, as a result of larger program sizes.

(3) The final cause that shall be mentioned here is the *duplication* of code in genetic programs. Code duplication may increase the amount of noneffective code in programs. This is much more likely with recombination by using existing genetic material from the population only. At least sequences of single identical instructions may be observed in linear genetic programs where only the last instruction can be effective. For single operations, e.g., $r_0 := r_1 + 2$, this is valid if the operand register(s) are different from the destination register. For noneffective duplications of instruction blocks all destination registers may not be used in later instructions of the same block, accordingly. The more registers are available the more likely this situation becomes.

9.9.3 Selection-Based Control

The simplest form of growth control in genetic programming is to choose the maximum size limit of programs as small as necessary for representing successful solutions (see Section 6.5). The problem is, however, that the optimum solution size is not known in advance. A popular approach to control program growth more implicitly is referred to as *parsimony pressure*. In contrast to a growth control over the variation operators (see previous section) a parsimony pressure is induced by means of selection. Usually this technique is implemented by integrating a size component into the fitness function that punishes longer programs by calculating a weighted sum of the two objectives *fitness* and *size* [51].

Following the principle of Occam's Razor a shorter solution can be expected better and more generic than a longer solution to the same problem. In general, parsimony pressure relies on the assumption that there is a positive correlation between shorter programs and better solutions. That is, solution finding profits from parsimony pressure when most good solutions are located in low-complex regions of the search space. Because such a correlation may not be assumed for each particular problem and each configuration of GP (see Section 6.5) parsimony pressure may not always be advantageous.

In the first place, the influence of parsimony pressure on the complexity and the evaluation time of linear genetic programs is interesting for the (structurally) effective code only. Recall that all structural introns can be removed efficiently from a linear genetic program and, thus, do not cause computational costs (see Section 3.2.1) during program execution in the fitness calculation or in the application domain.

In general, parsimony pressure is less important for the performace of linear GP. First, influence may be taken more directly on code growth over variation (parameters) than this is possible with a tree representation of programs. Reasons for this have been discussed in Sections 5.10.1 and 7.5. Basically, the higher variability of the linear representation has been held responsible for this which allows single instructions to be deleted or inserted freely at all program positions. Second, the presence of noneffective code, in general, already imposes an *implicit* parsimony pressure on the effective code in genetic programming. This is especially interesting when using crossover in linear GP (see also Chapter 7) where structural introns may be detected efficiently. Another important argument for using a variation-based growth control is that fitness selection is not disturbed.

9.9.4 Effective Complexity Selection

The separation of linear genetic programs in active code and inactive code on a structural level offers the possibility for a *code-specific* complexity control. This may be realized by a two-level tournament selection, a multi-objective selection method that has been introduced in Section 8.4. First, a certain number of individuals ($n > 2$) is selected by fitness and, second, among those only the two shortest programs are allowed to participate

Code	Selection	SSE		Length			Variations (%)		
	%	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
—	—	15.4	1.5	180	67	37	4.9	26	22
abs.	25	11.1	1.4	153	59	39	5.3	25	22
abs.	50	**9.6**	1.4	78	37	47	5.6	29	24
abs.	100	30.7	2.2	8	5	62	5.0	38	24
eff.	25	12.9	1.5	183	58	32	4.5	28	26
eff.	50	**12.2**	1.4	184	47	26	3.5	34	31
eff.	100	14.9	1.4	181	27	15	1.7	51	50
noneff.	25	10.9	1.4	149	64	43	5.7	24	21
noneff.	50	**9.4**	1.3	95	54	57	6.5	24	19
noneff.	100	19.3	2.1	51	45	88	7.0	26	16

Table 9.9: *mexican hat*: Second-level selection for effective, noneffective, and absolute complexity with selection rates 100%, 50%, 25% with crossover (cross). Average results over 100 runs after 1000 generations.

in variation. In order to limit code growth we may put a specific selection pressure on the individuals by choosing the smallest *effective*, *noneffective*, or *absolute* program length on the second level. Selection pressure is controlled by a selection probability that determines how often the complexity selection is applied.

Code-specific parsimony pressure has been proposed by Soule *et al.* [84] as a mean to restrict the growth of programs without restricting their effective code. The authors identified introns partly in tree programs as non-executed subtrees. These introns were induced by nested branches whose contradicting conditions were relatively easy to identify for a control problem. Recall that a relevant detection of introns in tree-based GP strongly depends on the program functions.

Experimental results in Tables 9.9 and 9.10 show for two test problems, *mexican hat* and *spiral*, that noneffective complexity selection is more successful than effective complexity selection when using unrestricted linear crossover. *Mexican hat* profits slightly from the latter variant, probably due to a stronger correlation between shorter programs and better solutions. This is in contrast to the *spiral* problem which is not better solved by an effective complexity selection at all. By imposing a specific pressure on the effective size the actual solution size is punished more specifically while the growth of noneffective code is almost not affected (see Tables 9.9 and 9.10). Thus, a smaller proportion of effective instructions is maintained in programs that reduces the effective crossover step size, but may increase the proportion of noneffective and neutral variations.

In both test cases a moderate punishment of the noneffective complexity has a positive influence on the prediction performance, by comparison. This is true even if effective step size becomes larger if the proportion of effective code increases, i.e., the rate of introns decreases. Instead, the proportion of noneffective and neutral variations becomes smaller. In Table 9.9 the absolute length is relatively more reduced than the effective length the higher this selection pressure is adjusted. In Table 9.10, instead, the effective size increases while the absolute size remains more-or-less unaffected. While in the first case the performance becomes worse, in the latter case the loss of structural introns is compensated by semantic introns. A similar effect has been obtained by removing the noneffective code completely during effective crossover (see Section 5.9.1) which corresponds more-or-less to a 100 percent selection for smallest noneffective code here.

Code	Selection	CE		Length			Variations (%)		
	%	*mean*	*std.*	*abs.*	*eff.*	%	*constr.*	*neutral*	*noneff.*
—	—	26.1	0.7	185	102	55	3.6	23	14
abs.	25	22.7	0.7	167	102	61	4.1	21	12
abs.	50	**20.9**	0.7	132	92	69	4.8	19	10
abs.	100	32.4	1.0	30	25	83	6.3	18	10
eff.	25	26.5	0.7	188	78	42	3.2	26	21
eff.	50	**26.0**	0.6	185	66	36	2.9	29	24
eff.	100	27.3	0.7	184	43	24	1.7	40	37
noneff.	25	**22.3**	0.7	179	134	75	4.1	20	8
noneff.	50	22.6	0.7	172	160	93	4.1	19	3
noneff.	100	23.1	0.7	182	181	99	3.5	20	1

Table 9.10: *spiral*: Second-level selection for effective, noneffective, and absolute complexity with selection rates 100%, 50%, 25% with crossover (cross). Average results over 100 runs after 1000 generations.

A code-specific complexity selection also allows us to investigate how much a selection pressure on the absolute length is depending on the reduction of effective code or noneffective code. If a general pressure works better than any code-specific pressure, the specific forms might complement each other. Unfortunately, prediction performance with an absolute complexity selection is hardly different from the results obtained with a noneffective complexity selection. This is true at least for moderate selection probabilities of 25 or 50 percent here. An absolute complexity selection produces smaller (effective) programs, however. At least when using crossover it prevents the semantic introns (in the structurally effective code) from growing as a protection against destructive variation effects.

A more reliable and stronger reduction of crossover step size on the effective code may be obtained by explicit introns (see Section 5.7.6). Those replace most noneffective instructions and, thus, reduce side-effects by reactivations. As a result, smaller effective solutions are possible. The reader may recall that EDIs constitute another method for controlling growth of effective code by means of selection.

One advantage of the two-level selection process over punishing the program length over a weighted term in the fitness function is that the primarily selection by fitness is less influenced. The two-level selection process better considers that fitness selection is prior to complexity selection. Furthermore, the selection pressure is easier to handle. Including multi-objective goals into the fitness requires an appropriate weighting of the objective terms to be found. Another problem of a (constant) weighting is that the pressure is stronger at the end of a run than at the beginning where programs are small. A second-level selection for complexity puts a more uniform pressure on the individuals that is more independent from their actual program length, but regards the relative differences in length.

Another variant of parsimony pressure, that is often applied in GP, selects the smaller individual only if two compared individuals share the same fitness. Obviously, with this method the selection pressure depends on the number of neutral fitness comparisons that occur with a problem and a system configuration (function set). Therefore, discrete fitness functions might be more affected than continuous fitness functions.

9.10 Conclusion

This chapter was about the phenomenon of code growth in genetic programming. Different reasons for code growth were investigated for the linear GP approach.

(1) We analyzed the influence of different variation effects on program size for different genetic operators in linear GP. In general, neutral variations were identified as a major cause of code growth and the emergence of introns. Almost no code bloat occurred if neutral variations are not accepted *and* if the structural step size of variations is reduced to a minimum. Both conditions make sure that intron instructions may not be created directly at the variation point. Recall that the linear (imperative) representation of programs allows structural variation steps to be constantly small. In general, the meaning of neutral variations is emphasized as a motor of evolutionary progress and code growth.

(2) We also reported on implicit length biases for some variation operators. In general, a relevant influence of the identified biases on the growth of genetic programs is doubtful. In particular, the removal bias theory could not be confirmed for linear crossover. Instead, an implicit shrink bias was detected with effective instruction mutations.

(3) Different methods for controlling code growth by variation or selection were presented. Recombination has been found to increase the size of programs much more dramatically than mutations in linear GP, especially if the variation step size is unrestricted for both macro operators. Several possible reasons were discussed to explain this phenomenon. Actually, code growth was affected only partly by the step size of macro mutations. Moreover, the two-level selection method from Chapter 8 was applied for a selective control of effective or noneffective program complexity.

In general, the following measures have proven to reduce the growth of code in linear GP, independently from their influence on the performance.

- ☐ Using macro mutation instead of recombination
- ☐ Reduction of variation step size
- ☐ Avoidance of neutral variations
- ☐ Avoiding a direct creation of neutral code (also by non-neutral variations)
- ☐ Implicit or explicit shrink bias in the variation operator
- ☐ (Effective) complexity selection

Chapter 10

Evolution of Program Teams

Contents

This chapter applies linear GP for the evolution of teams to several prediction problems including both classifications and regressions. Different linear methods for combining the outputs of the team programs are compared. These include hybrid approaches where (1) a neural network is used to optimize the weights of programs in a team for a common decision and (2) a real-numbered vector (the representation of evolution strategies) of weights is evolved with each team in parallel. The cooperative team approach results in an improved training and generalization performance compared to the standard GP method.

10.1 Introduction

Two main approaches can be distinguished concerning the combination of individual solutions in genetic programming: Either the individuals (genetic programs) can be evolved independently in different runs and combined *after* evolution, or a certain number of individuals can be *coevolved* in parallel as a *team*. The focus of this chapter is on the second approach.

Team evolution is motivated strongly by natural evolution. Many predators, e.g., lions, have learned to hunt pray in a pack most successfully. By doing so, they have developed cooperative behavior that offers them a much better chance to survive than single fellows. In GP the parallel evolution of team programs is expected to solve certain tasks more efficiently than the usual evolution of individuals. To achieve this the individual members of a team may solve the overall task in cooperation by specializing in subtasks for a certain degree.

Post-evolutionary combination, instead, suffers from the drawback that successful compositions of programs are detected randomly only. That might require a lot of runs to develop a sufficient number of individual solutions and a lot of trails to find a successful combination. Coevolution of k programs, instead, will turn out to be more efficient in time than k independent runs. Teams with highly cooperating and specialized members are hard to find by random especially since those usually require only a certain adaptation of their members to the training data. Most combinations of too much adapted (best-of-a-run) individuals may reduce the noise but may hardly develop cooperation.

Team solutions require the multiple decisions of their members to be merged into a collective decision. Several methods to combine the outputs of team programs are compared in this work. The coevolutionary team approach not only allows the combined error to be optimized but also an optimal *composition* of the programs to be found. In general, the optimal team composition is different from simply taking individual programs that are already quite perfect predictors for themselves. Moreover, the diversity of the individual decisions of a team may become an object of optimization.

In this chapter we also present a combination of GP and neural networks, the weighting of multiple team programs by a linear neural network (NN). The neural optimization of weights may result in an improved performance compared to standard combination methods. Recall that the name *linear* GP refers to the linear structure of the genetic programs. It does not mean that the method itself is linear, i.e., may solve linear separable problems only, as this is valid for linear NN. On the contrary, prediction models developed by GP may be highly non-linear.

In another hybrid approach the representations of linear GP and evolution strategies (ES) [91] are coevolved in that a vector of programs (team) and a vector of program weights form one individual and undergo evolution and fitness calculation simultaneously.

10.2 Team Evolution

Haynes *et al.* [38] introduced the idea of team evolution into the field of genetic programming. Since then evolution of teams has been investigated mostly in connection with cooperating agents solving multi-agent control problems. Luke and Spector [57] tested teamwork of homogeneous and heterogeneous agent teams in a predator/prey domain and showed that the heterogeneous approach is superior. In contrast to heterogeneous teams homogeneous teams are composed of completely identical agents and can be evolved with the standard GP approach. Haynes and Sen [39] tested a similar problem with different recombination operators for heterogeneous teams.

Preliminary studies about using a team approach for classifications appeared in [25] from the author of this work. Concurrently, Soule [88] applied teams to another non-control problem – a parity problem – by using majority voting to combine the Boolean member outputs. He [89] also documented specialization in teams for a linear regression problem and found better performance with teams when using a special voting method but not with averaging.

In this thesis the team approach is applied to three different prediction problems, two classification tasks and one approximation task. In data mining the generalization quality of predictive models, i.e., genetic programs here, is the most important criterion. In contrast to control tasks only heterogenous teams are of interest here, because for prediction tasks there is nothing to be gained from the combination of the outputs of completely identical programs (homogeneous teams).

10.2.1 Team Representation

In general, teams of individuals can be implemented in different ways. Firstly, a certain number of individuals can be selected randomly from the population and evaluated in combination as a team. The problem with this approach is known as the *credit assignment problem*: The combined fitness value of the team has to be shared and distributed among the team members.

Secondly, team members can be evolved in separate subpopulations which provide a more specialized development. In this case, the composition and the evaluation of teams might be separated from the evolution of their members by simply taking the best individuals from each deme in each generation and combining them. However, this raises another problem: An optimal team is not necessarily composed of best individuals for each team position. Specialization and coordination of the team's individuals is not a matter of evolution there. These phenomena might only emerge accidentally.

The third approach, favored here, is to use an explicit team representation that is considered as one individual by the evolutionary algorithm [39]. The population is subdivided into fixed, equal-sized groups of individuals. Each program is assigned a fixed position index in its team (program vector). The members of a team undergo a coevolutionary process because they are always selected, evaluated and varied simultaneously. This eliminates the credit assignment problem and renders the composition of teams an object of evolution.

Figure 10.1 shows the partitioning of the total population used in the experiments described below. First, the population is subdivided into *demes* [94] which, in turn, are subdivided into *teams* of individual programs. Exchange of genetic information between demes has not been realized by migration of whole teams. Instead, teams (tournament winners) are selected for recombination occasionally from *different* demes while their offspring

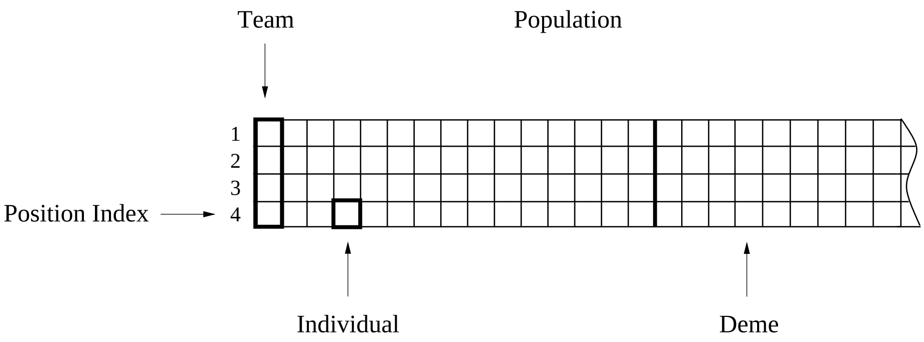

Figure 10.1: Population subdivided into teams and demes.

inherit code from both demes (*interdemetic recombination*). Demes are used because they better preserve the diversity of a population. This, in turn, reduces the probability of the evolutionary process to get stuck in a local minimum.

The coevolutionary approach prohibits teams of arbitrary size because the complexity of the search space and the training time, respectively, grow exponentially with the number of coevolved programs. On the other hand, the team size has to be large enough to cause an improved prediction compared to the traditional approach, i.e., team size one. Our experimental experience with this trade-off suggests that moderate numbers of team members are adequate (see Section 10.5).

10.2.2 Team Operators

Team representations require special genetic operators, notably for recombination. Genetic operations on teams, in general, reduce to the respective operations on their members which can be selected randomly. Researches [39] found that a moderate number of crossover points works better than recombining either one or every team position per operation. This is due to the trade-off between a sufficient variation, i.e., speed of the evolutionary process, and the destructive effect of changing too many team members at the same time.

For recombination the participating individuals of the two parent teams can be chosen of arbitrary or equal position. If recombination between team positions is forbidden completely, the members of a team evolve independently in isolated “member demes”. Luke and Spector [57] showed for a control problem that team recombination restricted in this way can outperform free recombination. Isolated or semi-isolated coevolution of the team members is argued to promote specialization in behavior.

A possible alternative to a random selection might be genetic operators that modify the team members depending on their respective individual fitness. Members may be sorted by error and the probability that an individual becomes a subject of mutation or crossover depends on its error rank. By doing so, worse member individuals are varied more often than better ones. Improving the fitness of worse members might have a better chance to improve the overall fitness of the team. However, we will see that there is not necessarily a positive correlation between a better member fitness and a better team fitness (see Section 10.5). Also note that this technique does not allow the member errors to differ much in a team which might have a negative effect on specialization, too.

10.3 Combination of Multiple Predictors

In principle, this chapter integrates two research topics, the evolution of teams discussed above and the combination of multiple predictors, i.e., classifiers or regressors. In contrast to teams of agents, teams whose members solve a prediction problem require the aggregation of the member's output to produce a common decision.

In the neural network community different approaches have been investigated dealing with the combination of multiple decisions in neural network *ensembles* [36, 73, 53]. Usually, neural networks are combined after training and are hence already quite perfect in solving a classification or approximation problem on their own. The ensemble members are not trained in combination and the composition of the ensemble does not undergo an optimization process. In [102] neural networks are evolved and a subset of the final population is combined afterwards. Different combination methods – including averaging and majority voting – are compared while a genetic algorithm is used to search for a near optimal ensemble composition.

For genetic programming Zhang *et al.* [104] applied a weighted majority algorithm in classification to combine the Boolean outputs of a selected subpopulation of genetic programs after evolution. This approach resulted in an improvement in generalization performance, i.e., robustness, compared to standard GP and simple majority voting, especially with sparse and noisy training data.

The decisions of different *types* of classifiers including neural networks and genetic programs are combined by an averaging technique in [83]. The result is an improved prediction quality of thyroid normal and thyroid carcinoma classes that has been achieved in a medical application.

10.3.1 Making Multiple Decisions Differ

In principle, all members in a team of predictors are intended to solve the same full task. The problem is not artificially subdivided among the members and there are no subproblems assigned to special team positions explicitly. In many real-world applications such subdivision would not be possible because the problem structure is completely unknown. We are interested in teams where specialization, i.e., a partitioning of the solution, emerges from the evolutionary process itself.

Specialization strongly depends on the heterogeneity of the teams. Heterogeneity is achieved by evolving members that produce slightly diverging outputs for the same input situations. Nothing will be gained from the combination of the outputs of completely identical predictors (homologous teams) as far as the quality of the solutions is concerned. Note that this is in contrast to agent teams that solve a control task where each agent program usually has side effects on the problem environment.

In genetic programming the inherent noise of the evolutionary algorithm already provides a certain heterogeneity of the team members. Additionally, it can be advantageous to restrict recombination between *different* team positions [57]. This is especially true if a team member does not "see" the full problem and is facing a more-or-less completely different subtask than the other members.

Otherwise, allowing *interpositional recombination* of teams allows innovative code to spread to the other positions in the team. Moreover, this exchange of genetic information between the "member demes" helps to better preserve the diversity of the overall team population. We will see in Section 10.5.3 that for teams of predictors an interpositional exchange of code does not necessarily reduce specialization potential and quality of results.

Besides restricted recombination there are more specific techniques to increase heterogeneity in teams and, thus, to promote the evolution of specialization:

One possible approach is to force the individuals of a team to disagree on decisions and to specialize in different domains of the training data. This can be achieved by either using different fitness functions for each member or by training each member with (slightly) different data sets. Both techniques require the individual errors of the members to be integrated into the fitness function (see Section 10.4.2). Otherwise, the effect of the different input situations cannot be made known to the evolutionary algorithm. Note that only member outputs of equal input situations can be used to calculate the combined error of the team.

Different training subsets for the team members can be derived from the full data set that is used to determine the training error of the team. For instance, small non-overlapping subsets may be left out as done with *cross validation*, a method used to improve the generalization capabilities of neural networks over multiple runs. The subsets may be sampled either at the beginning of run or resampled after a certain number of generations. The latter technique (*stochastic sampling*) introduces some additional noise in the sampling process. This may allow smaller and more different subsets to be used for the individual members since it guarantees that every team position over time is confronted with every training example.

Finally, different function sets can be chosen for different team positions to promote specialization as well. If recombination between different positions is allowed the team crossover operator has to be adapted in a way that only individual members built from the same function set are allowed to be recombined.

10.3.2 Combination Methods

The problem that arises with the evolution of team predictors is in the combination of the outputs of the individual members during fitness evaluation of a team. Different *combination methods* have been tested here. All methods compute the resulting team output from a *linear combination* of its member's outputs. Non-linear combination methods cannot necessarily be expected to produce better aggregations of multiple predictions since the actual problem, linear or non-linear, is already solved by the GP predictors. Figure 10.2 illustrates the general principle of the approach.

Moreover, only basic combination methods are documented and compared in this chapter. Even if there are hybridizations of the methods possible, e.g., EVOL/OPT or EVOL/MV (weighted majority voting), the concurrent application of two combinations is not necessarily more successful. We noticed that more complicated combination schemes are rather difficult to handle for the evolutionary algorithm. These might be more reasonable with post-evolutionary combinations of (independent) predictors. Most of the methods – except WTA (see Section 10.3.2) – can be applied to parallel as well as to sequentially evolved programs

For classification problems there exist two major possibilities to combine the outputs of multiple predictors: Either the raw output values or the classification decisions can be aggregated. In the latter case the team members act as full (pre)classifiers themselves. The drawback is that the mapping of the continuous outputs to discrete class identifiers *before* they are combined reduces the information content that each individual might contribute to the common team decision. Therefore, we decided for the former and combined raw outputs – except for majority voting (see below) that requires class decisions implicitly.

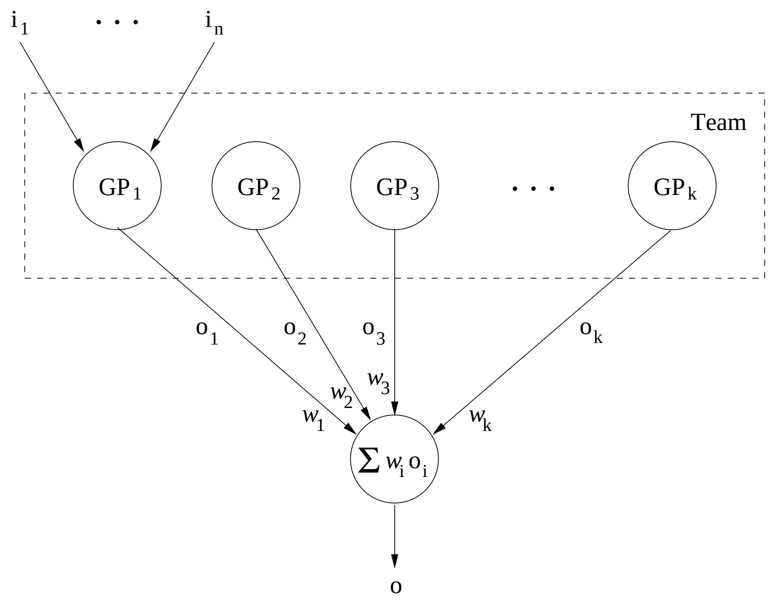

Figure 10.2: Linear combination of genetic programs.

Some of the combination methods are only applicable to classification tasks and are based upon one of the following two *classification methods*:

- ☐ *Classification with intervals* (INT). Each output class of the problem definition corresponds to a certain interval of the full value range of the (single) program output. In particular, for classification problems with two output classes the continuous program output is mapped to class output 0 or 1 here – depending on a classification threshold of 0.5. More generally, the class identifier is selected that is closest to the program output.

- ☐ *Winner-takes-all classification* (WTA). Here for *each* output class exactly one program output (output register) is necessary. The output with the highest value determines the class decision of the individual.

The following combination methods are introduced for problems with two output classes while a generalization to more output classes is not complicated. Even more important is to note that none of the methods presented here produces relevant extra computational costs.

Averaging (AV)

There are different variants of combination possible by computing a weighted sum of the outputs of the team programs. The simplest form is to use uniform weights for all members, i.e., the *simple average* of k outputs as team output. In this way the influence of each individual on the team decision is exactly the same. The evolutionary algorithm has to adapt the team members to the fixed weighting only.

$$o_{team} = \sum_{i=1}^{k} \frac{1}{k} o_{ind_i} \tag{10.1}$$

Weighting by Error (ERR)

An extended method is to use the fitness information of each team member for the computation of its weight. By doing so, better individuals get a higher influence on the team output than worse.

$$w_i = 1/e^{\beta E(gp_i)}. \tag{10.2}$$

$E(gp_i)$ is the individual error explained in Equation 10.9. β is a positive scaling factor to control the relation of the weight sizes. The error-based weighting gives lower weights to worse team members and higher weights to better ones. In order to restrict their range the weights always undergo normalization in that they are all positive and sum to one:

$$w_i = \left\| \frac{w_i}{\sum_{j=1}^{k} w_j} \right\| \tag{10.3}$$

With this approach evolution decides over the weights of a program member by manipulating its error value. In our experiments the individual weights are adjusted during training using the fitness information. Using data different from the training data may reduce overtraining of teams and increase their generalization performance. It has, however, the drawback of increasing computation time.

In general, the error-based weighting approach has not been found to be always better than the simple average of member outputs (see Section 10.5). The reason might be that the quality of a single member solution must not be directly related to the fitness of the whole team. If the combined programs had been evolved in single independent runs, deriving the member weights from this independent fitness might be a better choice. In such a case stronger dependences between programs – that usually emerge during team evolution by specialization – cannot be expected.

Coevolution of Weights (EVOL)

With this approach member weights are evolved in parallel with every team in the population (see Figure 10.3). The real-valued vector of weights is selected together with the vector of programs (team) by tournament selection. During each fitness evaluation the weight vector is varied by a certain number of mutations. In doing so, only better mutations are allowed to change the current state of weighting, a method typical for an (1+1)ES [91]. The mutation operator updates single weight values by allowing a constant standard deviation (*mutation step size*) of 0.02. The initial weights are randomly selected from the interval $[0, 1]$.

Alternatively, a complete (1+1)ES run might be initiated to optimize the weighting of each team during fitness calculation. This, of course, increases the computational costs significantly depending on the run length. It also might not be necessarily advantageous since the program teams adapt to a given weighting situation concurrently. With our approach optimization of the weighting is happening in coevolution with the members, not during each team evaluation. Thus, the coevolutionary aspect that allows team solutions to adapt to different weighting situations is the most important point here. Even if the diversity of the population decreases at the end of a GP run there are still improvements possible by changing the influences of the single team members.

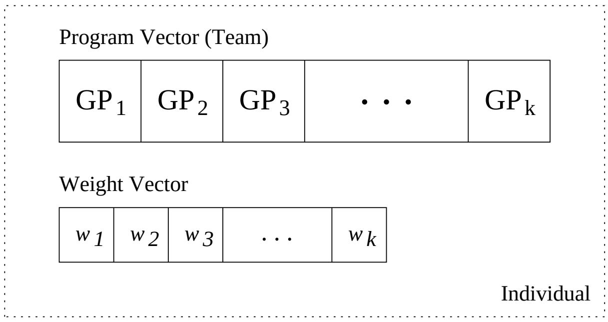

Figure 10.3: Coevolution of program team and vector of weights as individual.

Majority Voting (MV)

A special form of linear combination is *majority voting* which operates on *class* outputs. In other words, the continuous outputs of team programs are transformed into discrete class decisions *before* they are combined.

Let us assume that there are exactly two output classes, 0 and 1. Let O_c denote the subset of team members that predict class c:

$$O_0 := \{i | o_{ind_i} = 0, i = 1, .., k\} \tag{10.4}$$

$$O_1 := \{i | o_{ind_i} = 1, i = 1, .., k\} \tag{10.5}$$

The class which most of the individuals predict for a given example is selected as team output:

$$o_{team} = \begin{cases} 0 & : \quad |O_1| < |O_0| \\ 1 & : \quad |O_1| \geq |O_0| \end{cases} \tag{10.6}$$

Note that clear team decisions are forced for two output classes if an uneven number of members participates. Majority voting also works with an even number of members as long as the team decision is defined for equality (class 1 here).

Weighted Voting (WV)

Another voting method, *weighted voting*, is introduced here for the winner-takes-all classification (see above) where each team program returns exactly one output value for each of m output classes. For all classes c these values are summed to form the respective outputs of the team:

$$o_{team,c} = \sum_{i=1}^{k} o_{ind_i,c} \forall c \in \{0, .., m\} \tag{10.7}$$

The class with the highest output value defines the response class of the team as illustrated in figure 10.4.

With this combination method each team individual contributes a continuous "weight" for each class instead of a clear class decision. If discrete (class) outputs would be used the method corresponds to majority voting. Here the weighting comes from the member programs themselves. When using interval classification instead of WTA classification each program might compute its own weight in a separate (second) output variable.

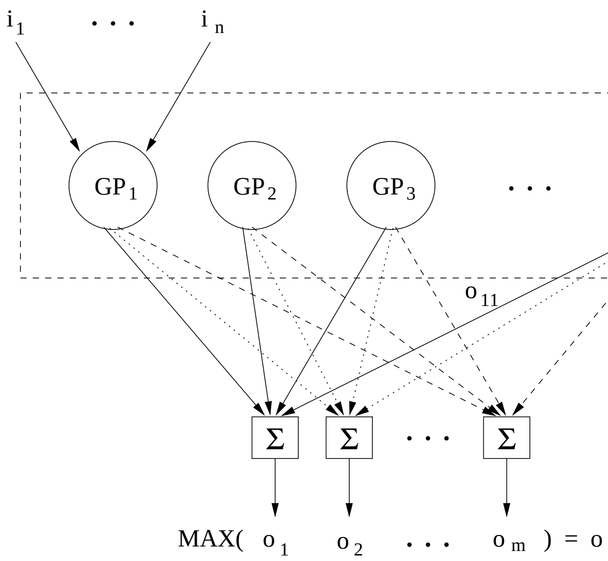

Figure 10.4: Combination of genetic programs by weighted voting.

Winner-Takes-All (WTA)

Two different *winner-takes-all combination* methods are distinguished: The first WTA combination variant selects the individual with the *clearest class decision* to determine the output of a team. With interval classification the member output that is closest to one of the class numbers (0 or 1) is identified as the clearest decision. The winner may also be seen as the individual with the highest *confidence* in its decision. Specialization may emerge if different members of the team win this contest for different fitness cases.

If separate outputs are used instead of output intervals (WTA *classification*) the clearest decision might be defined as the biggest difference between the highest output and the second highest output of a team member.

The second and simplest WTA combination (referred to as WTA2) just chooses the *minimum output* as team output. (Note that this is by definition and could be the maximum output as well.) This selection happens *before* the continuous outputs are transformed into class decisions and is valid for interval classification. For WTA classification the member with the lowest sum of outputs could be chosen. This combination variant is also possible for regression problems.

Of course, it is not a feasible alternative to select the member which output is closest to the desired output during training. Then a decision on unknown data is only possible if the right outputs are known in advance and is not made by the team itself.

Weight Optimization (OPT)

The final approach tested here uses a *linear* neural network in form of a perceptron *without* hidden nodes to find an optimal weighting of the team individuals. The learning method applied is RPROP [76], a backpropagation variant about as fast as Quickprop but with less adjustments of the parameters necessary. With this approach data is processed first by the team programs before the neural network combines their results (see also Figure 10.2). Actually, only a single neuron weights the connections to the genetic programs whose outputs represent the input layer of the linear neural network here. The outputs of the programs are, of course, only computed once for all data inputs before the neural weighting starts. In general, a predictor is trained using the outputs of multiple other predictors as inputs [99].

Like with the other approaches the neural weighting might be done each time the fitness of a team is calculated. Obviously, this has the drawback of an enormous increase in runtime even with a small neural network and a relatively low number of epochs trained. A much less time-consuming variant, that has been practiced here, is to apply weighting by average (AV) and to use the neural network only for optimizing the weights of the currently *best* team (outside of the population). By doing so, the process of finding an optimum weighting for the members is decoupled from the contrary process of breeding team individuals with a more balanced share in cooperation. By applying the neural weighting to *all* teams during evolution, instead, worse members may easily be "weighted out" of a team just by assigning them very low weights.

We compare only *linear* combination methods for the following reasons: First, non-linear combination of already non-linear predictors (genetic programs) will not necessarily result in better performance. Second, a non-linear combinator might solve too much of the prediction problem itself. The linear network structure assures that there is only a weighting of program outputs possible by the neural network and that the actual, non-linear problem is solved exclusively by the genetic programs. The neural combinator has been applied here for optimization because weighting is an inherent property of neural networks. Actually, using a non-linear (multi-layer) perceptron for the combination of the team programs instead did not produce significantly different results here than the linear aggregation. Moreover, the genetic programs stayed quite small (only a few effective instructions) and could hardly be regarded as a stand-alone team of predictors evolved by genetic programming.

10.4 Experimental Setup

We examine the team approach with different combination methods discussed earlier using two classification problems and one regression problem. First of all, the structure of the data that represents the respective problems is documented in further detail.

10.4.1 Structure of Experimental Data

The *heart* data set is composed of four data sets from the UCI Machine Learning Repository (*Cleveland, Hungary,* and *Switzerland*) and includes 720 examples altogether. The input dimension is 13 while two output classes (1 or 0) indicate the diagnosis (ill or not ill). The heart problem incorporates noise because inputs – including continuous and discrete values – are missing and have been completed with 0. The diagnosis task of the problem is to predict whether the diameter of at least one of four major heart vessel is reduced by more than 50 percent or not.

Two chains denotes a popular machine learning problem where two chained rings that represent two different classes – of 500 data points each – have to be separated. The two rings in Figure 10.5 "touch" each other at two regions without intersection.

The regression problem *three functions* tests the ability of teams to approximate three different functions at the same time which are a sine, a logarithm and a half circle (see Figure 10.6). 200 data examples were sampled for each function within input range $[0, 2\pi]$. A function index has to be passed to the genetic programs as an additional input to distinguish the three functions.

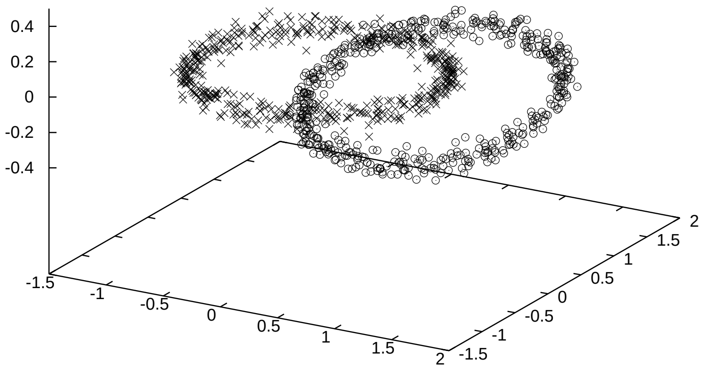

Figure 10.5: *two chains* problem.

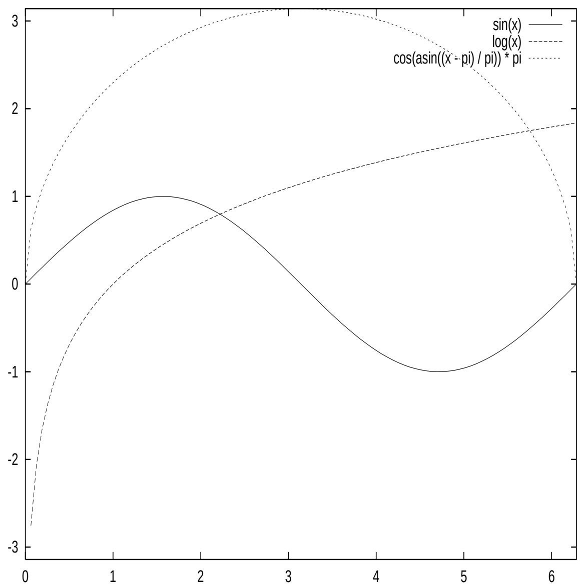

Figure 10.6: *three functions* problem.

The data examples of each problem were subdivided randomly into three sets: training set (50%), validation set (25%) and test set (25%). Each time a new best team occurs its error is calculated using the validation set in order to check its generalization ability *during* training. From all these best teams emerging over a run the one with minimum validation error is tested on the test set once *after* the training is over.

10.4.2 Team and Member Fitness

The *fitness* $\mathcal{F}$ of a team might integrate two major goals: the overall error of the team $E(team)$ and (optionally) the errors of its program members $E(gp_j)$ can be minimized.

$$\mathcal{F}(team) = \mathrm{E}(team) + \delta \cdot \frac{1}{m} \sum_{j=1}^{m} \mathrm{E}(gp_j) \tag{10.8}$$

In our experiments the combined team error and the member errors are both calculated for the complete training data. Provided that the outputs of the team members are saved the member errors are computed with almost no additional overhead.

The influence of the average member error on team fitness is controlled by a multiplicative parameter δ. Including the individual errors as a second fitness objective (by choosing $\delta = 1$) has not been observed to produce better results (see Section 10.5.3). If one wants to use different training sets for the different team positions (see Section 10.3.1), however, fitness shares of members are absolutely necessary. Note that the combined output of the team is computed for equal member inputs.

In Equation 10.8 E denotes the error of a predictor gp that is computed as the sum of square distances (SSE) between the predicted output(s) $gp(\vec{i_k})$ and the desired output(s) $\vec{o_k}$ over n examples $(\vec{i_k}, \vec{o_k})$:

$$\mathrm{E}(gp) = \sum_{k=1}^{n} (gp(\vec{i_k}) - \vec{o_k})^2 + w \cdot \mathrm{CE} = \mathrm{SSE} + w \cdot \mathrm{CE} \tag{10.9}$$

The *classification error* (CE) is calculated as the number of incorrectly classified examples in Equation 10.9. The influence of the classification error is controlled by a weight factor w. For classification problems w has been set constantly to 2 in order to favor classification quality (0 otherwise).

10.4.3 Parameter Settings

Table 10.1 lists the parameter settings of our linear GP system used for all experiments and problem definitions described above. The population size is 3000 teams while each team is composed of the same number of individual members. The population has been chosen sufficiently large to conserve diversity of the more complex team solutions. The total *number of members per team* and the *number of members that are varied* during crossover and mutation are the most important parameters when investigating the evolution of teams. Different settings of these parameters are reported in further detail in the next section.

Parameter	Setting
Number of generations	1000
Number of teams (population size)	3000
Number of team members	4
Number of varied team members	1–2
Number of demes	6
Interdemetic crossover	3%
Crossover probability	100%
Mutation probability	100%
Mutation step size for constants	5
Instruction set	$\{+, -, \times, /, x^y\}$
Set of (integer) constants	{0,..,99}
Maximum member length	128
Maximum initial member length	32

Table 10.1: General parameter settings.

The number of generations is limited to 1000, both for GP teams and standard GP. Note that member individuals are varied much less – one or two per team only – than stand-alone individuals. While this may reduce the progress speed of single team members it does not necessarily hold for the fitness progress of the whole team as we will see below.

A team is always varied simultaneously by crossover *and* mutation in our configuration. Mutations are only applied to member positions that have been changed during recombination.

10.5 Experimental Results

We now document the results obtained by applying the different team approaches described in 10.3.2 to the three problems of Section 10.4.1. Prediction accuracies and code sizes are compared for the team configurations and a standard GP approach.

The team approach, in general, has been found to produce better results than the standard GP approach for all three prediction tasks. First, mainly problems profit from a team evolution that may be divided at least partly into simpler subproblems that may be distributed among different problem solvers (team members). Only then team members may specialize and solve the overall task more successfully in cooperation.

Second, team solutions can be expected less brittle and more general in the presence of noisy training data. Due to their collective decision making the effect of noise may be reduced significantly. This functionality is true, however, already for combinations of stand-alone solutions.

If nearly optimal solutions already occur with the standard approach teams cannot be expected to be beneficial. In this case the additional computational overhead of the more complex team solutions outweighs the possible advantages.

10.5.1 Prediction Accuracy

Table 10.2 summarizes the different team approaches that have been discussed in Section 10.3.2. The outputs of the team members are continuous except for majority voting (MV) where the raw outputs have to be mapped on discrete class identifiers first. Only our weighted voting approach (WV) is based on the WTA classification method. All other methods use interval classification.

Method	Config.	Combination	Classification	Outputs
GP	—	—	INT	cont
TeamGP	AV	AVeraging (standard)	INT	cont
	OPT	weight OPTimization	INT	cont
	ERR	weighting by ERRor	INT	cont
	EVOL	coEVOLution of weights	INT	cont
	MV	Majority Voting	INT	class
	WV	Weighted Voting	WTA	cont
	WTA	Winner-Takes-All	INT	cont
	WTA2	Winner-Takes-All	INT	cont

Table 10.2: Configuration of the different team approaches.

The following tables compare best results of standard GP and the different team approaches for the three test problems introduced in Section 10.4. Minimum training error and minimum validation error are determined among best solutions (concerning fitness) of a run. The solution with minimum validation error is applied to unknown data at the end of a run to compute the test error. All figures given below denote average results from series of 60 test runs. In order to avoid unfair initial conditions and to give more reliable results each test series (configuration) has been performed with the same set of 60 random seeds.

Considering the classification rates for the *two chains* problem in Table 10.3 already the *standard team approach* (AV) reaches approximately an eight times better training performance than standard GP. Most interesting are the results of the winner-takes-all combination that select a *single* member program to decide for the team on a certain input situation. Both team variants (WTA and WTA2) nearly always found the optimum (0% CE) for training data and validation data. With standard GP the optimum solution has not even been found once during 60 trials here. This is a strong indication of a high specialization of the team members. It demonstrates clearly that highly coordinated behavior emerges from the parallel evolution of programs. This cannot be achieved by a combination of standard GP programs which have been evolved independently. Team evolution is much more sophisticated than just testing random compositions of programs. In fact, the different members in a team have adapted strongly to each other during the coevolutionary process.

Method	Training CE (%)	Member CE (%)	Validation CE (%)	Test CE (%)
GP	3.67 (0.25)	—	5.07 (0.30)	5.69 (0.37)
AV	0.44 (0.08)	25.8 (1.96)	0.82 (0.12)	2.08 (0.14)
OPT	0.36 (0.07)	32.1 (0.71)	0.69 (0.09)	1.96 (0.15)
ERR	1.31 (0.15)	20.9 (1.49)	1.91 (0.20)	2.73 (0.18)
EVOL	0.33 (0.07)	28.0 (2.09)	0.71 (0.16)	2.00 (0.17)
MV	0.37 (0.08)	25.7 (1.51)	1.48 (0.17)	2.17 (0.19)
WV	0.39 (0.09)	27.7 (1.98)	0.76 (0.14)	1.91 (0.18)
WTA	0.02 (0.01)	59.2 (2.27)	**0.00** (0.00)	**0.33** (0.18)
WTA2	**0.00** (0.00)	64.3 (1.53)	0.00 (0.00)	0.65 (0.29)

Table 10.3: *two chains*: Classification error (CE) in percent, averaged over 60 runs. Statistical standard error in parentheses.

Among the "real" team approaches which combine outputs of *several* individual members WV turned out to be about as successful as OPT and EVOL. This is remarkable because the WV method requires twice as many output values – two instead of one output per member – to be coordinated. Furthermore, the optimization of weights is coming from the member programs themselves within this variant.

Table 10.4 shows the prediction results for the *heart* problem. This application demonstrates not only the ability of teams in real data-mining but also in noisy problem environments since many data attributes are missing or are unknown. The difference in prediction error between GP and TeamGP is about 2 percent which is significant in the respective real problem domain. The problem structure does not offer many possibilities for specialization, especially in case of the winner-takes-all approaches which do not generalize significantly better here than the standard approach. The main benefit of the other combination methods seems to be that they improve fitness and generalization quality for

Method	Training CE (%)	Member CE (%)	Validation CE (%)	Test CE (%)
GP	13.6 (0.16)	—	14.5 (0.17)	19.0 (0.36)
AV	11.5 (0.15)	28.1 (2.18)	13.4 (0.18)	18.2 (0.30)
OPT	11.5 (0.17)	32.0 (2.03)	12.8 (0.18)	**17.5** (0.26)
ERR	11.9 (0.12)	28.6 (1.79)	12.9 (0.13)	18.0 (0.25)
EVOL	11.4 (0.13)	32.9 (2.39)	**12.7** (0.13)	18.1 (0.28)
MV	**10.9** (0.13)	24.6 (1.34)	13.6 (0.16)	17.5 (0.23)
WV	11.5 (0.11)	32.4 (2.41)	12.9 (0.15)	17.9 (0.24)
WTA	11.9 (0.17)	60.5 (2.44)	14.5 (0.22)	18.5 (0.31)
WTA2	12.9 (0.16)	61.5 (2.27)	14.9 (0.26)	19.2 (0.32)

Table 10.4: *heart*: Classification error (CE) in percent, averaged over 60 runs. Statistical standard error in parentheses.

the noisy data by a *collective* decision making of *more than one* team program.

Experimental results for the *three functions* problem are given in Table 10.5. Note that not all team variants are applicable to a regression problem. The regression task at hand has been solved most successfully by EVOL teams. This combination variant allows different weighting situations to be coevolved with the program teams and results in smaller prediction errors compared to uniform weights (AV). The standard team approach is found to be about four times better in training and generalization than the standard GP approach. Note that the average member error can become extremely high compared to the respective team error with this problem.

Method	Training MSE	Member MSE	Validation MSE	Test MSE
GP	16.9 (0.90)	—	16.2 (0.98)	16.6 (0.99)
AV	4.7 (0.27)	738 (50)	3.9 (0.22)	4.3 (0.25)
OPT	4.4 (0.30)	913 (69)	3.7 (0.27)	3.8 (0.27)
ERR	4.6 (0.33)	6340838 (4030041)	3.9 (0.30)	4.0 (0.30)
EVOL	**3.2** (0.27)	33135 (11041)	**2.6** (0.22)	**2.7** (0.24)
WTA2	11.0 (0.68)	154762629 (9025326)	9.8 (0.68)	10.1 (0.68)

Table 10.5: *three functions*: Mean square error (MSE $\times$ 100), averaged over 60 runs. Statistical standard error in parentheses.

Finally, some general conclusions can be drawn from the three applications:

Teams of predictors have proven to give superior results for known data as well as unknown data. On the one hand, specialization of team members has been held responsible for this. On the other hand, the improved generalization performance of teams may results from the increased robustness of team solutions against noise in the data space. This, in turn, is mainly due to the combination of multiple predictions that absorb ("smooth") larger errors or wrong decisions made by single members.

Comparing the different team configurations among each other further shows that different combination methods dominate for different problems. A general ranking of the methods cannot be produced. It is worth trying several variants when dealing with the evolution of multiple predictors.

Some methods that allow various weighting situations outperformed the standard team approach using uniform weights (AV). Among those methods the parallel evolution of weights together with the team programs (EVOL) turned out to be most successful. Optimizing the weights by using a neural network (OPT), instead, is done independently from evolution here (see Section 10.3.2). Because the individuals in best teams are already quite adapted to a fixed (uniform) weighting, optimization cannot be expected to lead to the same significant improvements.

For all three examples the average member error was highest with winner-takes-all combinations. This is not surprising since only one member is selected to make a final decision for the whole team while outputs of the other team individuals could be arbitrarily worse (WTA) or higher (WTA2) respectively. Apparently, specialization potential is highest with this combinations. In general, the member performance in teams is significantly worse than the performance of stand-alone GP individuals.

10.5.2 Code Size

The computational costs of team evolution (as compared to individual evolution) can be paid, at least in part, by the savings obtained from the following two effects:

- ☐ Only the (structurally) effective code is executed.
- ☐ The average effective code size of team members is significantly smaller than the effective size of stand-alone individual solutions.

As explained in Chapter 3 the (structurally) noneffective code is not executed and, thus, does not cause any computational costs no matter how complex it might become during the evolutionary process. The second effect is demonstrated in this section by comparing effective code sizes for different team configurations and standard GP. If no parsimony pressure is used, there is no selection pressure on the noneffective part of code. As a result, the absolute program size may grow almost unbounded and is limited only by the maximum size (number of members $\times$ 128 instructions here).

For the three example cases Tables 10.7, 10.6, and 10.8 show the effective and absolute code size of the best solutions. All teams hold the same number of members (4 here). WV combination that is based on winner-takes-all classification produces the largest teams. It seems that the multiple outputs calculated by WV members increase their complexity. WTA teams are found to be smallest in code size. Actually, they are not much bigger than a single standard individual in effective size and might even become smaller (see Table 10.6). This might be seen as another indication for the high specialization potential of the members in those teams. Among the other variants teams with non-uniform weights, like EVOL, are often found smaller than standard teams (AV). In general, concerning effective size teams become only about twice as big as standard individuals. For the heart problem they are not even 50 percent bigger. That means that, on average, a single member solution is definitely smaller than an individual solution.

The rates of noneffective code are comparably high for all team approaches. The intron rates of individual GP solutions are lower mostly because of a (relatively) higher restriction by the maximum size limit.

The average code size of teams in the population (not documented) has developed quite similar to the code size of best teams (averaged over multiple runs). Only for the *two chains* problem is the average size of WTA teams bigger. Note again that only the difference in average effective size of teams corresponds directly to the increase in runtime, when using intron elimination in linear GP (see Section 3.2.1).

Method	Code Size	Effective Size	Introns (%)
GP	128	45	64.8
AV	347	86	75.2
OPT	332	76	77.1
ERR	320	78	75.6
EVOL	294	67	77.2
MV	451	99	78.0
WW	448	124	72.3
WTA	92	**33**	64.1
WTA2	98	33	66.3

Table 10.6: *two chains*: Absolute and effective code size of teams with 4 members and standard GP in instructions. Effective code of teams about twice as big as standard individuals on average. WTA solutions are smaller than standard individuals.

One reason for the reduced growth of the (effective) team members could be seen in the *lower variation probability* compared to standard GP individuals. We will see in the following Section 10.5.3 that it is not recommended to vary too many members concurrently during a team operation. Best team prediction is obtained by varying about one member only. If only one team member is changed the probability for crossover at a certain team position is reduced by a factor equal to the number of members. One might conclude that member programs grow faster the more members are varied. That this is not true will be demonstrated in the experiments documented in Table 10.11 and 10.12 further below. Members with the best prediction accuracy and the biggest effective length occur with the *lowest* variation rate.

Method	Code Size	Effective Size	Introns (%)
GP	128	38	70.3
AV	488	56	88.5
OPT	485	48	90.1
ERR	479	46	90.3
EVOL	481	**44**	90.9
MV	497	56	88.7
WV	504	68	86.5
WTA	479	57	88.1
WTA2	405	48	88.1

Table 10.7: *heart*: Absolute and effective code size of teams with 4 members and standard GP in instructions. Effective code of teams not even 50 percent bigger than standard individuals on average.

As a result, there must be another reason than variation speed for the relatively small (effective) size of teams. We have already seen in the last section that teams perform better than standard individuals after a sufficient number of generations. In order to make team solutions more efficient there must be *cooperations* occurring between the members that specialize to solve certain subtasks. These subtasks can be expected to be less difficult than the main problem wherefore the respective subsolutions are more likely less complex in effective size than a full one-program solution. Conclusively, a positive correlation between smaller (effective) member size and higher degree of specialization might be supposed.

Method	Code Size	Effective Size	Introns (%)
GP	128	58	54.7
AV	435	131	69.9
OPT	432	125	71.1
ERR	465	136	70.8
EVOL	456	123	73.0
WTA2	354	**76**	78.5

Table 10.8: *three function*: Absolute and effective code size of teams with 4 members and standard GP in instructions.

10.5.3 Parameter Analyses

In this section we analyze the influence of the most relevant parameters when dealing with the evolution of program teams. First of all, those are the number of team members (team size) and the number of members that are selected from a team during a genetic operation. Both prediction errors and code sizes are compared for various settings of these parameters. Beyond that, two further parameters are under consideration that are of interest in this context: the influence of free recombination between member positions and the individual member errors on the fitness. In the preceding experiments recombination was restricted to equal positions exclusively while the individual errors were not regarded (see Section 10.4.2).

Instead of giving a detailed analysis for each team variant and each test problem, we restrict the following experiments to the standard team approach (AV). Combination by simple average has the advantage that each member solution has exactly the same influence on the team decision. This makes teams with a single dominating member less likely. Even if experiments are not documented for all problems very similar results have been observed with the other prediction tasks.

Number of Team Members

Each team member is varied by crossover or mutation with a probability of 50 percent in order to guarantee a comparison as fair as possible. Modifying only one member at a time, for instance, would be unfair since then the variation speed of members reduces directly with their number. But, on the other hand, the more members are varied at the same time the more difficult it becomes to make small improvements to the combined team output.

Table 10.9 compares the classification errors (CE) for the *two chains* problem and different numbers of team members ranging from one (standard GP) to eight. Using teams with more individuals might be rather computationally unacceptable even though only effective instructions are executed in our GP system. Both prediction performance and generalization performance increase with the number of members. But from a team size of about 4 members significant improvements do not occur any more.

The correlation between the number of members and the average code size of a member (in number of instructions) is shown in Table 10.10. The maximum code size of each member is restricted to 128 instructions. The absolute size and the effective size per

#Members	Training CE (%)	Member CE (%)	Validation CE (%)	Test CE (%)
1	3.33 (0.31)	3.3 (0.31)	4.70 (0.35)	5.59 (0.39)
2	1.33 (0.21)	16.5 (1.23)	2.34 (0.33)	3.31 (0.31)
3	0.89 (0.17)	23.1 (1.89)	1.59 (0.27)	2.64 (0.28)
4	0.37 (0.06)	27.4 (1.91)	0.69 (0.12)	1.84 (0.20)
5	0.36 (0.08)	32.8 (1.53)	0.47 (0.12)	1.90 (0.17)
6	0.38 (0.08)	32.6 (2.01)	0.58 (0.11)	1.76 (0.16)
7	0.30 (0.06)	30.2 (2.35)	0.48 (0.10)	1.78 (0.16)
8	0.39 (0.09)	34.1 (2.32)	0.48 (0.09)	1.76 (0.11)

Table 10.9: *two chains*: Classification error (CE) in percent for different number of team members, averaged over 60 runs. Statistical standard error in parentheses. Half of the team members are varied.

#Members	Member Size	Effective Size	Introns (%)
1	128	46	64.0
2	126	36	71.4
3	98	25	74.5
4	94	20	78.7
5	82	19	76.8
6	85	21	75.3
7	75	18	76.0
8	73	18	75.3

Table 10.10: *two chains*: Average member size in instructions for different numbers of team members. Half of the team members are varied.

member decrease up to team size 4 here. Beyond that, both sizes stay almost the same. This corresponds directly to the development in prediction quality from Table 10.9. Note that the amount of genetic material of the whole team still increases with the number of members.

The reason for the reduction in effective member size can be seen in a distribution of the problem task among the team individuals whereby the subtask each member has to fulfill gets smaller and easier. A second indication for that might be the average member error that has been calculated for the full training set here. As shown in Table 10.9 the error increases respectively. Probably, beyond a certain number of individuals the task cannot be split more efficiently so that some members must fulfill more-or-less the same. As a result, members keep to a certain effective size and prediction quality.

The intron rate is not affected significantly even though genetic operators change more members (always 50 percent) simultaneously in bigger teams. Only with very few members this rate is lower. But this is due to the maximum size limit that restricts mainly the growth of intron code. The otherwise rather constant rate of noneffective code (and effective code respectively) can be explained by the influence of each member on the team output that decreases with the total number of members – especially if uniform member weights are used. As a result, the intervention of crossover should be almost the same here for all configurations (in contrast to Table 10.11) and higher protection by more introns is not needed. Moreover, this is also an explanation of why team errors in Table 10.9 do not get worse again from a certain number of individuals.

Number of Varied Members

As stated above best results occur when only a moderate number of team members, i.e., one or two, is varied simultaneously by crossover or mutation. This is demonstrated in Table 10.11 where the number of varied members ranges from 1 to a maximum of 4 while the team size stays fixed. Prediction and generalization performance are found best if only one individual is varied at a time.

Table 10.12 demonstrates the correlation between the number of varied team members and the code size of teams. Interestingly, effective and absolute code size reduce with the variation strength. Although the variation probability per member is lowest if only one member is varied during a team operation the (effective) code is biggest. Concurrently, the overall prediction accuracy of teams increases while the (average) member error is highest with the lowest level of variation in Table 10.11. Some reasons can be found to explain these phenomena:

#Varied Members	Training MSE	Member MSE	Validation MSE	Test MSE
1	4.1 (0.37)	903 (92)	3.4 (0.30)	3.7 (0.36)
2	5.4 (0.47)	730 (73)	4.8 (0.45)	4.9 (0.47)
3	6.5 (0.44)	538 (50)	5.5 (0.38)	6.3 (0.48)
4	8.3 (0.66)	421 (53)	7.1 (0.61)	7.6 (0.70)

Table 10.11: *three functions*: Mean square error (MSE $\times$ 100) with different numbers of varied members, averaged over 60 runs. Statistical standard error in parentheses. Number of team members is 4.

#Varied Members	Code Size	Effective Size	Introns (%)
1	440	148	66.4
2	424	125	70.5
3	388	113	70.9
4	320	99	69.1

Table 10.12: *three functions*: Code size of team in instructions for different numbers of varied members. Number of team members is 4.

One reason might be the fact that, in general, smaller steps in variation allow more directed improvements of a solution than bigger steps. In particular, single team individuals may specialize stronger within the collective. By doing so, their errors in relation to a solution of the overall task as well as their complexity increase. As already observed in Section 10.5.1 higher member errors correspond to a higher degree in specialization again.

On the other hand, the effect of variation on a team becomes more destructive the more members participate in it. Then it might be easier for smaller (effective) team solutions to survive during evolution. Decreasing complexity is the dominating protection mechanism here. The intron rate is not affected significantly, i.e., the proportion of effective and noneffective code stays rather constant. The reader may recall that similar results have been found in Section 5.11.4 such that smaller variation step sizes (numbers of mutation points) produced better and larger effective programs.

Interpositional Recombination

It has been argued in Section 10.3.1 that in teams of multiple predictors – where by definition each member solves the same problem – allowing recombination between different member positions might be more successful than restricting it to equal positions only (*intrapositional recombination*). Only by interpositional recombination member code can be moved from one position to another in the team.

Recombination	Training MSE	Member MSE	Validation MSE	Test MSE
free	0.34 (0.05)	25.7 (1.42)	0.65 (0.10)	1.82 (0.11)
restricted	0.44 (0.08)	25.8 (1.96)	0.82 (0.12)	2.08 (0.14)

Table 10.13: *two chains*: Classification error (CE) in percent, averaged over 60 runs, with restricted (reprinted from Table 10.3) and unrestricted recombination. Statistical standard error in parentheses.

Recombination	Training MSE	Member MSE	Validation MSE	Test MSE
free	4.4 (0.27)	682 (44)	3.7 (0.23)	3.8 (0.23)
restricted	4.7 (0.27)	738 (50)	3.9 (0.22)	4.3 (0.25)

Table 10.14: *three functions*: Mean square error (MSE $\times$ 100), averaged over 60 runs, with restricted (reprinted from Table 10.5) and unrestricted recombination. Statistical standard error in parentheses.

Tables 10.13 and 10.14 show results for restricted and unrestricted recombination when using combination by simple average (AV). Actually, free recombination performs slightly better than restricted recombination with the tested problems. At least, it does not seem to have any negative influence here. Thus, intrapositional recombination might be less relevant when dealing with teams of predictors. Experiments with other combination methods produced comparable results.

Member Fitness

δ	Training MSE	Member MSE	Validation MSE	Test MSE
0	0.44 (0.08)	25.8 (1.96)	0.82 (0.12)	2.08 (0.14)
1	1.91 (0.21)	12.4 (0.61)	3.00 (0.25)	3.92 (0.28)

Table 10.15: *two chains*: Classification error (CE) in percent, averaged over 60 runs, with and without including member fitness in Equation 10.8. Statistical standard error in parentheses.

δ	Training MSE	Member MSE	Validation MSE	Test MSE
0	4.7 (0.27)	738 (50)	3.9 (0.22)	4.3 (0.25)
1	19.4 (0.49)	34.6 (1.6)	18.0 (0.49)	18.1 (0.51)

Table 10.16: *three functions*: Mean square error (MSE $\times$ 100), averaged over 60 runs, with and without including member fitness. Statistical standard error in parentheses.

Finally, we investigate the effect of including ($\delta = 1$) or not including ($\delta = 0$) the average member error in the fitness function (Equation 10.8). Results documented in Tables 10.15 and 10.16 for weighting by average have been found to be representative for other combination methods, too. The average fitness of team members becomes significantly better. Actually, this reduces the specialization potential of members because the cooperating individuals are restricted to be good predictors on their own. As a result, the quality of team prediction decreases significantly if individual errors are included.

If, on the other hand, individual errors are not included in the fitness function there is no direct relation between fitness of a single member and the quality of the common team solution. This allows the errors of members to differ quite strongly within a team and to be significantly larger than the team error.

10.6 Combination of Multiple Program Outputs

In standard case, a single register content is defined as the output of a linear program. Apart from that, linear GP allows the program response to be derived from more than one or all registers. These outputs may be interpreted as multiple predictions of a single program solution and can be combined by using the same methods as proposed for team solutions in this chapter.

On the one hand, an aggregation of multiple outputs may be supposed to promote an internal parallelism of calculations as well as a specialization of subprograms. On the other hand, it has to be noted that a linear program may already combine multiple calculation paths, i.e., the contents of multiple registers, inside itself.

Depending on the number of registers provided by the user complementary subsolutions may be computed by using more-or-less independent sets of registers in the same program. These subprograms represent more-or-less disconnected components of the data flow graph (see Section 3.3). A complete disconnection as between team members, however, is rather unlikely, even if the number of registers is high (compared to the number of inputs).

Finally, the (effective) programs are probably larger when using multiple outputs than programs with a single output only. This is already true because registers will be effective for a higher probability. As a result, the speed advantage of evaluating a single program instead of multiple team members only would be relaxed, at least in part.

10.7 Discussion and Future Research

First of all, it is interesting to determine problem classes for which the team approach is suitable in general or for which it cannot produce better results than the standard approach.

The exchange of information between the individuals of a team might help to evolve a better coordinated behavior. One possibility in linear GP is, for instance, to share some calculation variables between team members that together implement a *collective memory.* Values can be assigned to these variables by one individual and used by others that are executed later on. Note that with using such a *shared memory* the evaluation order of the team members has to be observed. Another possible form of information sharing is the coevolution of submodules with each team that can be used by all its members in common (*shared submodules*).

Teams offer the possibility for an alternative parallelization approach in genetic programming that is different from distributing subpopulations of individuals to multiple processors. The member programs of a team can be executed in parallel by assigning each member to its own processing unit. If all members of the same position index ("member deme") belong to the same unit and interpositional recombination is not applied then migration of program code between processing nodes is not necessary. The only communication overhead between the units would be the exchange of team identifier and team outputs.

Finally, the numerous alternatives that have been given in the text may be a subject of future research.

A drawback of team solutions could be that they are probably more difficult to analyze than single genetic programs. But because already single solutions are often quite difficult to understand this might be a rather negligible disadvantage. Moreover, a combination of subsolutions can be more simple than a one-program solution as well.

10.8 Conclusion

The results of this chapter may be summarized as follows:

(1) The team approach was applied successfully to several prediction problems and has been found to reduce both training error and generalization error significantly compared to the individual approach. This was already achieved by using standard averaging to combine the outputs of the team programs.

(2) Several linear combination methods were compared while different methods turned out to be the most successful ones. Two benchmark problems were presented on which either a winner-takes-all combination (WTA) or the coevolution of variable member weights (EVOL) performed notably better than other approaches.

(3) The average effective complexity of teams with four members was only about two times larger than stand-alone solutions. With some combination methods team solutions have been found that are even smaller. Thus, the evolution of program teams is quite efficient provided that noneffective instructions are not executed.

(4) A high degree of specialization and cooperation has been observed such that team members showed a much lower prediction performance and a smaller (effective) size than individuals. Beyond a certain optimum number of team members, however, both features did not change anymore. One explanation could be that the overall problem task cannot be further divided into smaller subtasks.

(5) By including the prediction errors of members in the fitness function (of teams) their specialization potential may be reduced drastically. While the average member performance increased here the overall team performance decreased.

(6) The best team solutions emerged if not more than one team member is varied at a time. Interestingly, teams occured to be smaller and less highly specialized if several members are varied simultaneously.

Chapter 11

Summary and Outlook

This thesis reports on linear genetic programming (LGP), a variant of genetic programming (GP) that evolves computer programs as sequences of instructions of an imperative programming language. In general, the research focus is on basic structural aspects of the linear representation rather than on problem-specific or semantic aspects, like the evolved programming language. Fundamental differences to the traditional tree representation comprise the graph-based functional structure of linear genetic programs as well as the existence of structurally noneffective code, i.e., graph components that are not connected to the effective component or data flow. These structural aspects motivate in part the two major objectives of this thesis: (1) the development of advanced LGP methods and genetic operators to produce better and more compact program solutions and (2) the analysis of general EA/GP phenomena in the area of linear GP.

The first two chapters give an introduction to the general GP approach and into linear GP in particular. Chapter 3 shows how the special imperative representation of programs that is used in this thesis may be transformed into a directed acyclic graph (DAG). Thus, linear GP may be reduced to a special form of graph-based GP. Efficient algorithms are introduced for analyzing linear genetic programs in terms of certain features of their imperative structure or functional structure. Besides *structural introns*, these features comprise the *number of effective registers* at a program position, the *dependence degree* of (effective) instructions, and the *effective dependence distance*. Fundamental results of this chapter are published in [19, 13].

Especially, the elimination of noneffective code during runtime, which happens once before a program is executed repeatedly during the fitness calculation, may accelerate the processing time of GP significantly. Among other things, this is demonstrated in Chapter 4 and publication [19]. The proportion of noneffective code in programs depends on the genetic operators and on the configuration of various system parameters.

In Chapter 5 possible variation operators for the linear program representation are discussed and compared primarily in terms of their influence on prediction quality. In general, larger improvements in performance occurred in combination with a significant restriction of the maximum variation step size, either indirectly over a smaller solution size or directly over variation parameters. Noteworthy small or even minimum step sizes on the instruction level turned out to be optimum. In general, a linear (imperative) representation is more suitable to be varied in small steps than a tree structure. Due to its weaker constraints and the possibility of structural code deactivations, variation step sizes may be permanently small at each program position. Additionally, the efficiency of variations is enhanced significantly by increasing the proportion of (structurally) effective and/or (semantically) neutral variations. To achieve this, information about the program structure and/or about the program behavior need to be integrated into the variation process. A combination of both strategies leads to effective instruction mutations which performed most successfully. This approach appeared in [13].

Moreover, variation-specific parameters are analysed in Chapter 5 together with the corresponding variation operators. More general control parameters of linear GP are the subject of Chapter 6. For instance, the number of registers in imperative genetic programs influences their functional structure, i.e., the maximum width and depth of the DAG as well as the degree of node connections.

Comparisons of linear GP with other methods are conducted in Chapters 4 and 7. In Chapter 4 the standard LGP approach is compared with backpropagation neural networks using the RPROP learning rule. On a collection of medical classification problems both approaches show a competitive generalization performance (see [19]).

In Chapter 7 we compare tree-based GP with different variants of linear GP in terms of both prediction quality and solution size. Two sets of benchmark problems have been composed. One includes artifical GP benchmarks while the other one includes classification problems from bioinformatics. Linear GP is superior, especially when applying the more sophisticated operators from Chapter 5. In general, a larger difference in performance has been found between maximum and minimum step sizes of linear genetic operators than between the two representation types (when applying unrestricted recombination). Moreover, (effective) linear genetic programs have been found more compact due to (1) a multiple usage of register contents and (2) an implicit parsimony pressure by the structurally noneffective code.

While in Chapter 5 minimum variation steps are investigated in terms of the absolute program structure, the step size on the effective program is minimized in Chapter 8. Therefore, this effective step size is quantified by means of a structural distance metric which is sufficiently correlated to the fitness distance. Best solutions emerge if not more than one effective instruction changes its effectiveness status after variation. That is, only a single node may be connected to or disconnected from the effective component in the DAG. Even without applying such an explicit control mechanism, the effective code develops increasingly robust against deactivations over the generations. That is, the frequency of effective step sizes decreases already implicitly to a certain extent. This effect is referred to as self-protection. Furthermore, noteworthy improvements in performance have been achieved by increasing the diversity, i.e., the structural distance between programs, in the population actively. Results of Chapter 8 may be found in [21, 13].

The phenomenon of code bloat in linear GP is in the center of interest in Chapter 9. Mostly by intron code genetic programs may grow larger than necessary without showing corresponding improvements in fitness. When using instruction mutations almost only neutral variations turned out to be responsible for both the creation and the propagation of introns in genetic programs. Actually, programs hardly grow if neutral variation effects are not accepted *and* if the step size of macro variations is minimum. In doing so, effective instruction mutations have been identified as a genetic operator with which programs grow hardly larger than necessary. Especially, the emergence of noneffective code is reduced significantly. Thus, this operator realizes an implicit complexity control in linear GP which reduces a possible negative effect of code growth to a minimum. Another interesting result is that the program size increases strongly with recombination while it is hardly influenced by mutation in linear GP even if the maximum step size is not explicitly restricted in both cases. The first part of Chapter 9 has been adopted from [22].

Most results presented in this thesis refer to genetic programs as linear sequences of imperative operations. Program teams are investigated as one possibility to enlarge the complexity and dimension of LGP solutions. Chapter 10 reflects results of contribution [20] and applies the team approach to several prediction tasks. This requires a definite way of how the multiple predictions of team members are combined. Depending on the problem, different combination methods proved to be the most successful ones. We demonstrate that much more powerful solutions are possible with a team representation than those that may be found by the evolution of individuals. Moreover, the effective complexity of teams is surprisingly small compared to individual solutions. Both is possible by a high degree of specialization and cooperation of the team members.

Future research may proceed in the following directions. These result from restrictions that have been imposed on the program representation or from initial conventions that have been made for this thesis.

(1) *Representation and Genetic Operators.* This thesis deals with linear genetic programs as sequences of operations and conditional operations. The expressiveness of programs may be increased by applying more advanced programming concepts. A general overview of possible concepts has been given in Chapter 2. Their capability may be verified especially for prediction problems that have been favored in this thesis. On the one hand, the evolved programming language may be enhanced, for instance, by conditional forward jumps or backward jumps (loops) over larger instruction blocks. This makes the linear order of instructions to differ more strongly from the execution order. Moreover, an analysis of (functional and imperative) program features may become more difficult and more computationally expensive. This is true, for instance, for the detection of structural introns because registers that are effective at a certain program position may change dynamically during multiple executions of code blocks. For the same reason, the proportion of structural introns may be expected smaller. Instead, the proportion of semantic introns may be larger if the execution of several instructions depends on the same condition.

On the other hand, the complexity of the (linear) representation may be increased by combining multiple instruction sequences (blocks) in a more-or-less predefined manner. Concerning the team approach from Chapter 10 this is a linear combination of member outputs. In [45] instruction sequences are connected by a branching graph structure (see Section 2.1.4). Such two-level program representations require appropriate two-level variation operators to be defined.

As argued before, the efficiency of a programming concept or a program representation strongly depends on the genetic operators. If a concept is not really needed for more successful solutions or if a profitable usage is rather unlikely during the automatic evolution, the resulting larger search space may influence solution finding rather negatively.

(2) *GP Phenomena.* Second-level program structures may develope different variants of structural (and semantic) introns which require more sophisticated detection algorithms and may contribute to the growth of programs in different ways.

(3) *Evolutionary Algorithm.* While this thesis concentrates on aspects of linear GP that are closely related to the representation of programs, other parts of the evolutionary algorithm have been kept unchanged. For instance, the selection method is always tournament selection in combination with a steady-state population. Other selection schemes and representation-independent EA parameters may be investigated in terms of their influence on the performance of linear GP and the particular methods that have been developed here.

(4) *Cross Analyses.* It would go beyond the scope of this thesis to test all interesting combinations of the parameter analyses, methods, and genetic operators that have been discussed. Hence, there are still interesting configurations left. For instance, the team approach could be applied together with effective instruction mutations. In general, the documented results may be verified for other test problems or configurations as those that have been used here.

Bibliography

[1] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, San Francisco, CA, 2002.

[2] L. Altenberg, *Emergent Phenomena in Genetic Programming.* In A.V. Sebald and L.J. Fogel (eds.) *Proceedings of the third Annual Conference on Evolutionary Programming*, pp. 233–241, World Scientific, 1994.

[3] L. Altenberg, *The Evolution of Evolvability in Genetic Programming.* In K.E. Kinnear (ed.) *Advances in Genetic Programming*, ch. 3, pp. 47–74, MIT Press, Cambridge, MA, 1994.

[4] D. Andre and J.R. Koza, *Parallel Genetic Programming: A Scalable Implementation Using The Transputer Network Architecture.* In P.J. Angeline and K.E. Kinnear (eds.) *Advances in Genetic Programming 2*, pp. 317–337, MIT Press, Cambridge, MA, 1996.

[5] P.J. Angeline and J.B. Pollack, *The Evolutionary Induction of Subroutines.* In *Proceedings of the Fourteenth Conference of the Cognitive Science Society*, Lawrence Erlbaum Associates, Hilldale, NJ, 1992.

[6] P.J. Angeline, *Two Self-Adaptive Crossover Operators for Genetic Programming.* In P.J. Angeline and K.E. Kinnear (eds.) *Advances in Genetic Programming 2*, ch. 5, pp. 89–110, MIT Press, Cambridge, MA, 1996.

[7] P.J. Angeline, *Subtree Crossover: Building Block Engine or Macro-mutation.* In J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba, and R.L. Riolo (eds.) *Genetic Programming 1997: Proceedings of the Second Annual Conference* (GP'97), pp. 9–17, Morgan Kaufmann, San Francisco, CA, 1997.

[8] T. Bäck, *Self-Adaptation.* In T. Bäck, D. B. Fogel, and Z. Michalewicz (eds.), *Handbook of Evolutionary Computation*, ch. C7.1, Oxford University Press, New York, 1997.

[9] W. Banzhaf, *Genetic Programming for Pedestrians.* In S. Forrest (ed.) *Proceedings of the Fifth International Conference on Genetic Algorithms* (ICGA'93), p. 628, Morgan Kaufmann, San Francisco, CA, 1993.

[10] W. Banzhaf, *Genotype-Phenotype-Mapping and Neutral Variation: A Case Study in genetic programming.* In Y. Davidor, H.-P. Schwefel, and R. Männer (eds.) *Parallel Problem Solving from Nature (PPSN) III*, pp. 322–332, Springer-Verlag, Berlin, 1994.

[11] W. Banzhaf, P. Nordin, R. Keller, and F. Francone, *Genetic Programming – An Introduction. On the Automatic Evolution of Computer Programs and its Application.* dpunkt/Morgan Kaufmann, Heidelberg/San Francisco, 1998.

[12] W. Banzhaf and W.B. Langdon, *Some Considerations on the Reason for Bloat.* Genetic Programming and Evolvable Machines, vol. 3(1), 81–91, 2002.

[13] W. Banzhaf, M. Brameier, M. Stautner, and K. Weinert. *Genetic Programming and its Application in Machining Technology.* In H.-P. Schwefel, I. Wegener, and K. Weinert (eds.) *Advances in Computational Intelligence – Theory and Practice*, Springer, Berlin, 2002.

[14] W.G. Baxt, *Applications of Artificial Neural Networks to Clinical Medicine.* Lancet, vol. 346, pp. 1135–1138, 1995.

[15] C.L. Blake and C.J. Merz, *UCI Repository of Machine Learning Databases* [`http://www.ics.uci.edu/~mlearn/MLRepository.html`]. University of California, Department of Information and Computer Science.

[16] T. Blickle and L. Thiele, *Genetic Programming and Redundancy.* In J. Hopf (ed.) *Genetic Algorithms within the Framework of Evolutionary Computation* (Workshop at KI-94), Technical Report No. MPI-I-94-241, pp. 33–38, Max-Planck-Institut für Informatik, 1994.

[17] T. Blickle and L. Thiele, *A Comparion of Selection Schemes Used in Genetic Algorithms.* Technical Report 11/2, TIK Institute, ETH, Swiss Federal Institute of Technology, 1995.

[18] M. Brameier, W. Kantschik, P. Dittrich, and W. Banzhaf, *SYSGP - A C++ Library of Different GP Variants.* Technical Report CI-98/48, Collaborative Research Center 531, University of Dortmund, 1998.

[19] M. Brameier and W. Banzhaf, *A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining.* IEEE Transactions on Evolutionary Computation, vol. 5(1), pp. 17–26, 2001.

[20] M. Brameier and W. Banzhaf, *Evolving Teams of Predictors with Linear Genetic Programming.* Genetic Programming and Evolvable Machines, vol. 2(4), pp. 381–407, 2001.

[21] M. Brameier and W. Banzhaf, *Explicit Control of Diversity and Effective Variation Distance in Linear Genetic Programming.* In J.A. Foster, E. Lutton, J. Miller, C. Ryan, and A.G.B. Tettamanzi (eds.) *Proceedings of the Fifth European Conference on Genetic Programming* (EuroGP 2002), LNCS 2278, pp. 37–49, Springer, Berlin, 2002. (best paper award)

[22] M. Brameier and W. Banzhaf, *Neutral Variations Cause Bloat in Linear GP.* In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa (eds.) *Proceedings of the Sixth European Conference on Genetic Programming* (EuroGP 2003), LNCS 2610, pp. 286–296, Springer, Berlin, 2003. (best poster paper award)

[23] P.A. Castillo, J. Gonzles, J.J. Merelo, A. Prieto, V. Rivas, and G. Romero, *SA-Prop: Optimization of Multilayer Perceptron Parameters using Simulated Annealing.*, 1999.

[24] K. Chellapilla, *Evolving Computer Programs without Subtree Crossover.* IEEE Transactions on Evolutionary Computation, vol. 1(3), pp. 209–216, 1998.

[25] Collaborative Research Center SFB 531, *Design and Management of Complex Technical Processes and Systems by Means of Computational Intelligence Methods*, Internal Report, University of Dortmund, 1999.

[26] N.L. Cramer, *A Representation for the Adaptive Generation of Simple Sequential Programs.* In J. Grefenstette (ed.) *Proceedings of the First International Conference on Genetic Algorithms* (ICGA'85), pp. 183–187, 1985.

[27] P. Dittrich, F. Liljeros, A. Soulier, and W. Banzhaf, *Spontaneous Group Formation in the Seceder Model.* Physical Review Letters, vol. 84, pp. 3205–3208, 2000.

[28] R. Floyd and R. Beigel, *The Language of Machines.* International Thomson Publishing, 1996.

[29] L.J. Fogel, A.J. Owens, and M.J. Walsh, *Artificial Intelligence through Simulated Evolution.* Wiley, New York, 1996.

[30] R. Friedberg, *A Learning Machine, Part I.* IBM Journal of Research and Development, vol. 2, pp. 2–13, 1958.

[31] R. Friedberg, B. Dunham, and J. North, *A Learning Machine, part II.* IBM Journal of Research and Development, vol. 3, pp. 282–287, 1959.

[32] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[33] H.F. Gray, R.J. Maxwell, I. Martinez-Perez, C. Arus, and S. Cerdan, *Genetic Programming for Classification of Brain Tumours from Nuclear Magnetic Resonance Biopsy Spectra.* In J.R. Koza, D.E. Goldberg, David B. Fogel, and Rick L. Riolo (eds.) *Genetic Programming 1996: Proceedings of the First Annual Conference* (GP'96), p. 424, MIT Press, Cambridge, MA, 1996.

[34] J.J. Grefenstette, *Predictive Models Using Fitness Distributions of Genetic Operators.* In L.D. Whitley and M.D. Vose (eds.) *Foundations of Genetic Algorithms 3*, pp. 139–161, Morgan Kaufmann, San Francisco, CA, 1995.

[35] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.

[36] L.K. Hansen and P. Salamon, *Neural Network Ensembles.* IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 12(10), pp. 993–1001, 1990.

[37] K. Harries and P. Smith, *Exploring Alternative Operators and Search Strategies in Genetic Programming.* In In J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba and R.L. Riolo (eds.) *Genetic Programming 1997: Proceedings of the Second Annual Conference* (GP'97), pp. 147–155, Morgan Kaufmann, San Francisco, CA, 1997.

[38] T. Haynes, S. Sen, D. Schoenefeld, and R. Wainwright, *Evolving a Team.* In *Working Notes for the AAAI Symposium on Genetic Programming*, MIT Press, Cambridge, MA, 1995.

[39] T. Haynes and S. Sen, *Crossover Operators for Evolving a Team.* In In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo (eds.) *Genetic Programming 1997: Proceedings of the Second Annual Conference* (GP'97), pp. 162–167, Morgan Kaufmann, San Francisco, CA, 1997.

[40] J. Holland, *Adaption in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, 1975.

[41] C. Igel and K. Chellapilla, *Investigating the Influence of Depth and Degree of Genotypic Change on Fitness in Genetic Programming.* In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, and R.E. Smith (eds.) *Proceedings of the International Conference on Genetic and Evolutionary Computation* (GECCO'99), pp. 1061–1068, Morgan Kaufmann, San Francisco, CA, 1999.

[42] C. Igel and K. Chellapilla, *Fitness distributions: Tools for designing efficient evolutionary computations.* In L. Spector, W.B. Langdon, U.-M. O'Reilly, and P.J. Angeline (eds.) *Advances in Genetic Programming III*, ch. 9, MIT Press, Cambridge, MA, 1999.

[43] T. Jones and S. Forrest, *Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms.* In L.J. Eshelmann (ed.), *Proceedings of the Sixth International Conference on Genetic Algorithms* (ICGA'95), pp. 184–192, Morgan Kaufmann, San Francisco, CA, 1995

[44] E.D. de Jong, R.A. Watson, and J.B. Pollack, *Reducing Bloat and Promoting Diversity using Multi-Objective Methods.* In L. Spector et al. (eds.), *Proceedings of the Third International Conference on Genetic and Evolutionary Computation* (GECCO 2001), pp. 11–18, Morgan Kaufmann, San Francisco, CA, 2001.

[45] W. Kantschik and W. Banzhaf, *Linear-Graph GP – A New GP Structure.* In J.A. Foster, E. Lutton, J. Miller, C. Ryan, and A.G.B. Tettamanzi (eds.) *Genetic Programming, Proceedings of the 5th European Conference* (EuroGP 2002), pp. 83–92, Springer-Verlag, LNCS, Berlin, 2002.

[46] R. Keller and W. Banzhaf, *Explicit Maintenance of Genetic Diversity on Genospaces*, Internal Report, University of Dortmund, 1995.

[47] M. Kimura and G.H. Weiss, *The Stepping Stone Model of Population Structure and the Decrease of Genetic Correlation with Distance.* Genetics, vol. 49, pp. 313–326, 1964.

[48] M. Kimura, *The Neutral Theory of Molecular Evolution.* Cambridge University Press, 1983.

[49] M. Kimura, *Some Recent Data Supporting the Neutral Theory.* In *New Aspects of the Genetics of Molecular Evolution*, Springer-Verlag, Berlin, 1991.

[50] J.R. Koza, *Hierarchical Genetic Algorithms Operating on Populations of Computer Programs.* In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pp. 768–774, Morgan Kaufmann, San Francisco, CA, 1989.

[51] J.R. Koza, *Genetic Programming – On the Programming of Computer Programs by Natural Selection.* MIT Press, Cambridge, MA, 1992.

[52] J.R. Koza, *Genetic Programming II – Automatic Discovery of Reusable Programs.* MIT Press, Cambridge, MA, 1994.

[53] A. Krogh and J. Vedelsby, *Neural Network Ensembles, Cross Validation, and Active Learning.* In G. Tesauro, D.S. Touretzky and T.K. Leen (eds.) *Advances in Neural Information Processing Systems*, vol. 7, pp. 231–238, MIT Press, Cambridge, MA, 1995.

[54] W.B. Langdon and R. Poli, *Fitness Causes Bloat.* In P.K. Chawdhry, R. Roy, and R.K. Pant (eds.) *Soft Computing in Engineering Design and Manufacturing*, pp. 13–22, Springer-Verlag, Berlin, 1997.

[55] W.B. Langdon, T. Soule, R. Poli, and J.A. Foster, *The Evolution of Size and Shape.* In L. Spector, W.B. Langdon, U.-M. O'Reilly, and P.J. Angeline (eds.) *Advances in Genetic Programming III*, pp. 163–190, MIT Press, Cambridge, MA, 1999.

[56] W.B. Langdon, *Size Fair and Homologous Tree Genetic Programming Crossovers.* Genetic Programming and Evolvable Machines, vol. 1:(1/2), pp. 95–119, 2000.

[57] S. Luke and L. Spector, *Evolving Teamwork and Coordination with Genetic Programming.* In J.R. Koza, D.E. Goldberg, David B. Fogel, and Rick L. Riolo (eds.) *Genetic Programming 1996: Proceedings of the First Annual Conference* (GP'96), pp. 150–156, MIT Press, Cambridge, MA, 1996.

[58] S. Luke and L. Spector, *A Revised Comparison of Crossover and Mutation in Genetic Programming.* In J.R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, D.E. Goldberg, H. Iba, and Rick Riolo (eds.) *Genetic Programming 1998: Proceedings of the Third Annual Conference* (GP'98), pp. 208–213, Morgan Kaufmann, San Francisco, CA, 1998.

[59] B. Manderick, M. de Weger, and P. Spiessens, *The Genetic Algorithm and the Structure of the Fitness Landscape.* In R. Belew and L. Booker (eds.), *Proceedings of the Fourth International Conference on Genetic Algorithms* (ICGA'91), pp. 143–150, Morgan Kaufmann, San Francisco, CA, 1991

[60] J.F. Miller and P. Thomson, *Cartesian Genetic Programming.* In R. Poli, W. Banzhaf, W.B. Langdon, J.F. Miller, P. Nordin, and T.C. Fogarty (eds.) *Genetic Programming, Proceedings of the 4th European Conference* (EuroGP 2000), pp. 121–132, Springer-Verlag, LNCS, Berlin, 2000.

[61] T. Mitchell, *Machine Learning.* McGraw-Hill, New York, 1996.

[62] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings – Practical Online Search Algorithms for Texts and Biological Sequences*, Cambridge University Press, 2002.

[63] P.S. Ngan, M.L. Wong, K.S. Leung, and J.C.Y. Cheng, *Using Grammar Based Genetic Programming for Data Mining of Medical Knowledge.* In J. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, D.E. Goldberg, H. Iba, and R.L. Riolo (eds.) *Genetic Programming 1998: Proceedings of the Third Annual Conference* (GP'98), Morgan Kaufmann, San Francisco, CA, 1998.

[64] P. Nordin, *A Compiling Genetic Programming System that Directly Manipulates the Machine-Code.* In K.E. Kinnear (ed.) *Advances in Genetic Programming*, pp. 311–331, MIT Press, Cambridge, MA, 1994.

[65] P. Nordin and W. Banzhaf, *Complexity Compression and Evolution.* In L.J. Eshelman (ed.) *Proceedings of the Sixth International Conference on Genetic Algorithms* (ICGA'95), pp. 310–317, Morgan Kaufmann, San Francisco, CA, 1995.

[66] P. Nordin and W. Banzhaf, *Evolving Turing-Complete Programs for a Register Machine with Self-Modifying Code.* In L. Eshelman (ed.) *Proceedings of Sixth International Conference of Genetic Algorithms* (ICGA'95), pp. 318–325, Morgan Kaufmann, San Francisco, CA, 1995.

[67] P. Nordin, F. Francone, and W. Banzhaf, *Explicit Defined Introns and Destructive Crossover in Genetic Programming.* In P. Angeline and K.E. Kinnear (eds.) *Advances in Genetic Programming II*, pp. 111–134, MIT Press, Cambridge, MA, 1996.

[68] P.J. Nordin, *Evolutionary Program Induction of Binary Machine Code and its Applications.* PhD thesis, University of Dortmund, Department of Computer Science, 1997.

[69] P. Nordin, W. Banzhaf, and F. Francone, *Efficient Evolution of Machine Code for CISC Architectures using Blocks and Homologous Crossover. Advances in Genetic Programming III*, pp. 275–299, MIT Press, Cambridge, MA, 1999

[70] U.M. O'Reilly and F. Oppacher, *Program Search with a Hierarchical Variable Length Representation: Genetic Programming, Simulated Annealing, and Hill Climbing.* In Y. Davidor, H.-P. Schwefel, and R. Männer (eds.) *Parallel Problem Solving from Nature (PPSN) III*, pp. 397–406, Springer-Verlag, Berlin, 1994.

[71] U.M. O'Reilly and F. Oppacher, *A Comparative Analysis of GP.* In P.J. Angeline and K.E. Kinnear (eds.) *Advances in Genetic Programming 2*, pp. 23–44, MIT Press, Cambridge, MA, 1996.

[72] U.-M. O'Reilly, *Using a Distance Metric on Genetic Programs to Understand Genetic Operators.* In J.R. Koza (ed.), *Late Breaking Papers at the Genetic Programming '97 Conference*, Standford University, 1997.

[73] M.P. Perrone and L.N. Cooper, *When Networks Disagree: Ensemble Methods for Neural Networks.* In R.J. Mammone (ed.) *Neural Network for Speech and Image Processing*, pp. 126–142, Chapman-Hall, London, 1993.

[74] L. Prechelt, PROBEN1 – *A Set of Neural Network Benchmark Problems and Benchmarking Rules.* Technical Report 21/94, University of Karlsruhe, 1994.

[75] I. Rechenberg, *Evolutionsstrategie '94.* Frommann-Holzboog, 1994.

[76] M. Riedmiller and H. Braun, *A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm.* In *Proceedings of the International Conference on Neural Networks* (ICNN'93), pp. 586–591, San Francisco, CA, 1993.

[77] B.D. Ripley and R.M. Ripley, *Neural Networks as Statistical Methods in Survival Analysis.* In R. Dybowski and V. Grant (eds.) *Artificial Neural Networks: Prospects for Medicine*, Landes Biosciences Publishers, Texas, 1997.

[78] J.P. Rosca and D.H. Ballard, *Causality in Genetic Programming.* In L.J. Eshelmann (ed.), *Proceedings of the Sixth International Conference on Genetic Algorithms* (ICGA'95), pp. 256–263, Morgan Kaufmann, San Francisco, CA, 1995

[79] J.P. Rosca, *Generality Versus Size in Genetic Programming.* In J.R. Koza, D.E. Goldberg, D.B. Fogel, and R.L. Riolo (eds.) *Genetic Programming 1996: Proceedings of the First Annual Conference* (GP'96), pp. 381–387, MIT Press, Cambridge, MA, 1996.

[80] D. Sankoff and J.B. Kruskal (eds.), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, MA, 1983.

[81] B. Sendhoff, M. Kreutz, and W. von Seelen, *A Condition for Genotype-Phenotype-Mapping: Causality.* In T. Bäck (ed.) *Proceedings of the Seventh International Conference on Genetic Algorithms* (ICGA'97), pp. 73-80, Morgan Kaufmann, San Francisco, CA, 1997.

[82] P.W.H. Smith and K. Harries, *Code Growth, Explicitly Defined Introns, and Alternative Selection Schemes.* Evolutionary Computation, vol. 6(4), pp. 339–360, 1998.

[83] R.L. Somorjai, A.E. Nikulin, N. Pizzi, D. Jackson, G. Scarth, B. Dolenko, H. Gordon, P. Russell, C.L. Lean, L. Delbridge, C.E. Mountford and I.C.P. Smith, *Computerized Consensus Diagnosis – A Classification Strategy for the Robust Analysis of MR Spectra. 1. Application to H-1 Spectra of Thyroid Neoplasma.* Magnetic Resonance in Medicine, vol. 33, pp. 257–263, 1995.

[84] T. Soule, J.A. Foster, and J. Dickinson, *Code Growth in Genetic Programming.* In J.R. Koza, D.E. Goldberg, D.B. Fogel, and R.L. Riolo (eds.) *Genetic Programming 1996: Proceedings of the First Annual Conference* (GP'96), pp. 215–223, MIT Press, Cambridge, MA, 1996.

[85] T. Soule and J.A. Foster, *Code Size and Depth Flows in Genetic Programming.* In J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba and R.L. Riolo (eds.) *Genetic Programming 1997: Proceedings of the Second Annual Conference* (GP'97), pp. 313–320, Morgan Kaufmann, San Francisco, CA, 1997.

[86] T. Soule and J.A. Foster, *Removal Bias: A new Cause of Code Growth in Tree-based Evolutionary Programming.* In *Proceedings of the International Conference on Evolutionary Computation* (ICEC'98), pp. 781–786, IEEE Press, New York, 1998.

[87] T. Soule and J.A. Foster, *Effects of Code Growth and Parsimony Pressure on Populations in Genetic Programming.* Evolutionary Computation, vol. 6(4), pp. 293–309, 1999.

[88] T. Soule, *Voting Teams: A Cooperative Approach to Non-Typical Problems using Genetic Programming.* In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, and R.E. Smith (eds.) *Proceedings of the International Conference on Genetic and Evolutionary Computation* (GECCO'99), pp. 916–922, Morgan Kaufmann, San Francisco, CA, 1999.

[89] T. Soule, *Heterogeneity and Specialization in Evolving Teams.* In Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer (eds.) *Proceedings of the Second International Conference on Genetic and Evolutionary Computation* (GECCO 2000), pp. 778–785, Morgan Kaufmann, San Francisco, CA, 2000.

[90] T. Soule and R.B. Heckendorn, *An Analysis od the Causes of Code Growth in Genetic Programming.* Genetic Programming and Evolvable Machines, vol. 3(3), pp. 283–309, 2002.

[91] H.-P. Schwefel, *Evolution and Optimum Seeking.* Wiley, New York, 1995.

[92] H.-P. Schwefel, I. Wegener, and K. Weinert (eds.) *Advances in Computational Intelligence – Theory and Practice*, Springer-Verlag, Berlin, 2002.

[93] A. Teller, *Turing Completeness in the Language of Genetic Programming with Indexed Memory.* In *Proceedings of the World Congress on Computational Intelligence* (WCCI'94), vol. 1, pp. 136-141, IEEE Press, New York, 1994.

[94] W.A. Tackett, *Recombination, Selection and the Genetic Construction of Computer Programs.* PhD thesis, University of Southern California, Department of Electrical Engineering Systems, 1994.

[95] R. Tanese, *Distributed Genetic Algorithms.* In J.D. Schaffer (ed.) *Proceedings of the Third International Conference on Genetic Algorithms* (ICGA'89), pp. 434–439, Morgan Kaufmann, San Francisco, CA, 1989.

[96] A. Teller, *PADO: A New Learning Architecture for Object Recognition.* In *Symbolic Visual Learning*, Oxford University Press, 1996.

[97] J.D. Watson, N.H. Hopkins, J.W. Roberts, J.A. Steitz, and A.M. Weiner, *Molecular Biology of the Gene.* Benjamin/Cummings Publishing Company, 1987.

[98] D. Wiesmann, *Anwendungsorientierter Entwurf evolutionärer Algorithmen.* Dissertation, Shaker Verlag, Aachen, 2001. (in German)

[99] D.H. Wolpert, *Stacked Generalization.* Neural Networks, vol. 5(2), pp. 241–260, 1992.

[100] D.H. Wolpert and W.G. Macready, *No Free Lunch Theorem for Optimization.* IEEE Transactions on Evolutionary Computation, vol. 1(1), pp. 67–82, 1997.

[101] S. Wright, *Isolation by Distance.* Genetics, vol. 28, pp. 114–138, 1943.

[102] X. Yao and Y. Liu, *Making Use of Population Information in Evolutionary Artificial Neural Networks.* IEEE Transactions on Systems, Man and Cybernetics, vol. 28B(3), pp. 417–425, 1998.

[103] T. Yu and J. Miller, *Neutrality and the Evolvability of Boolean Function Landscapes.* In J.F. Miller, M. Tomassini, P.L. Lanzi, C. Ryan, A.G.B. Tettamanzi and W.B. Langdon (eds.) *Genetic Programming, Proceedings of the 4th European Conference* (EuroGP 2001), pp. 204–217, Springer-Verlag, LNCS, Berlin, 2001.

[104] B.-T. Zhang and J.-G. Joung, *Enhancing Robustness of Genetic Programming at the Species Level.* In J.R. Koza, D.E. Goldberg, David B. Fogel, and Rick L. Riolo (eds.) *Genetic Programming 1996: Proceedings of the First Annual Conference* (GP'96), pp. 336–342, MIT Press, Cambridge, MA, 1996.