Evolutionary System Identification - Modern Concepts and Practical Applications

DISSERTATION

zur Erlangung des akademischen Grades

DOKTOR DER TECHNISCHEN WISSENSCHAFTEN

IM DOKTORATSSTUDIUM DER TECHNISCHEN WISSENSCHAFTEN

Angefertigt am *Institut für formale Modelle und Verifikation*

Eingereicht von:

Dipl.-Ing. Stephan M. Winkler

Betreuung:

Priv.-Doz. Dipl.-Ing. Dr. Michael Affenzeller

Beurteilung:

Priv.-Doz. Dipl.-Ing. Dr. Michael Affenzeller

Univ.-Prof. Dipl.-Ing. Dr. Luigi del Re

Linz, April 2008

Acknowledgements

Doing the work reported on in this thesis surely would not have been possible (or as enjoyable as it was) without the support of a number of people.

First of all, I have to thank my supervisor Dr. Michael Affenzeller for his continuous support over the last few years. It was him who introduced me to evolutionary optimization strategies, such as genetic algorithms and genetic programming, and also successfully organized the research project I have been working on over the last two years; he has always been available for questions, offered useful suggestions and substantially supported me in designing and testing the evolutionary system identification framework presented in this thesis.

The work described here would not have been possible in such a convenient way without being able to use the HeuristicLab framework. I am very grateful to Stefan Wagner who is the project manager and key developer of the HeuristicLab, not only for his effort on our software framework, but also for repeated design discussions and valuable hints on various topics. I would also like to thank all members of the Heuristic and Evolutionary Algorithms Laboratory (HEAL) for a very enjoyable way of cooperation in our research on evolutionary computation.

Furthermore, I would like to express my gratitude to Prof. Dr. Luigi del Re for giving me the opportunity to work at the Institute for Design and Control at JKU Linz, and also all colleagues at the institute. Special thanks go to Prof. del Re for his numerous advices and hints regarding heuristic system identification seen from a different point of view, and for being the second supervisor of this thesis. I am thankful to Prof. Dr. Witold Jacak, the dean of the School of Informatics, Communications and Media at Hagenberg, Austria for making it possible for me to work at the Research Center Hagenberg.

Finally, I have to thank my parents, my family, and all of my friends and colleagues who have encouraged me during my work on this thesis – and of course also Veronika for proofreading this thesis and supporting me in easy as well as in hard times.

Essentially, this thesis reflects on research work that has been supported by the Austrian Science Foundation (FWF) in project L284-N04, "GP-Based Techniques for the Design of Virtual Sensors". Parts of the work described here have been published in the articles listed in the publications section of this thesis; therefore I would like to thank the co-authors for helpful suggestions on certain sections and aspects, which also improved the presentation of the respective results in this thesis.

Abstract

Evolutionary computation is a subfield of computational intelligence that uses concepts inspired by natural evolution: Solutions to a given problem are represented by individuals of a population of solution candidates, and these individuals evolve iteratively by repeated selection of parents for producing new individuals. One of the most famous evolutionary techniques is the genetic algorithm, a global optimization technique using aspects inspired by evolutionary biology such as selection, recombination, mutation and inheritance. Genetic programming (GP) is an extension to the genetic algorithm that is able to automatically search for computer programs that solve given problems.

In principle, system identification denotes the generation of mathematical models for systems based not on a priori knowledge, but rather on measured data; the result of a system identification algorithm consists in a mathematical description of the behavior of the analyzed system. In this thesis we concentrate on system identification techniques based on genetic programming: Mathematical expressions are produced by an evolutionary process that uses the given measurement data.

The first part of this thesis summarizes the theoretical concepts used, starting with a summary of basic principles of genetic algorithms, genetic programming and data based modeling consisting of structure identification and parameter optimization. The GP based structure identification method implemented as a set of plug-ins for the HeuristicLab framework is described in detail, as well as a series of concepts that can be used for monitoring population dynamics during the execution of the GP process; we concentrate on genetic diversity and aspects of genetic propagation. The application of further developed selection principles and additional optimization stages is also explained as well as on-line and sliding window GP variants.

The second part of this thesis summarizes the results of empirical system identification test series that were executed using the GP concepts described in the first part. The data sets used here include dynamic measurement data of technical, mechatronical systems as well as classification benchmark problems taken from the UCI machine learning repository. The results of these test series do not only demonstrate the ability of this method to produce models of high quality for different kinds of machine learning problems, but also give insights into population dynamic processes that occur during the execution of a genetic programming process.

The present thesis is completed by a conclusion summarizing the results presented, a bibliography and the author's CV.

Kurzfassung

Evolutionäre Algorithmen sind im Allgemeinen Algorithmen, deren Verhalten durch Vorgänge der natürlichen Evolution inspiriert ist: Lösungen für ein gegebenes Problem werden durch Individuen in Populationen von Lösungskandidaten repräsentiert, und im Laufe eines iterativen, evolutionären Prozesses entwickeln sich diese Lösungskandidaten weiter durch die wiederholte Produktion von neuen Lösungskandidaten basierend auf der genetischen Information von zuvor selektierten Elternindividuen. Einer der bekanntesten Vertreter dieser Klasse von Algorithmen ist der genetische Algorithmus, eine globale Optimierungsstrategie welche Aspekte der natürlichen Evolution (Selektion, Rekombination, Mutation und Vererbung) benützt. Genetische Programmierung (GP) ist eine Erweiterung des genetischen Algorithmus welche in der Lage ist, automatisiert Computerprogramme zu generieren, um ein gegebenes Problem zu lösen.

Unter Systemidentifikation versteht man allgemein die datenbasierte Generierung von mathematischen Modellen, um das Verhalten von System zu beschreiben. Die Grundlage der Systemidentifikation besteht dabei rein aus den gegebenen Meßdaten; das Resultat dieser Modellgenerierung ist ein mathematischer Ausdruck, welcher das Verhalten des zu untersuchenden Systems repräsentieren soll. Das Hauptaugenmerk dieser Arbeit liegt auf Systemidentifikation basierend auf genetischer Programmierung: Die Generierung eines mathematischen Ausdrucks geschieht durch einen evolutionären Prozeß, welcher die gegebenen Meßdaten als Grundlage verwendet.

Der erste Teil dieser Arbeit befaßt sich hauptsächlich mit den theoretischen Konzepten welche in diesem Kontext relevant sind. Die Grundlagen genetischer Algorithmen sowie genetischer Programmierung werden ebenso erläutert wie allgemein datenbasierte Modellierung bestehend aus Strukturidentifikation und Parameteroptimierung. Darauf folgt eine detaillierte Erklärung, wie GP-basierte Strukturidentifikation als eine Reihe von Plugins für das HeuristicLab Framework implementiert wurde. Eine Reihe von Konzepten zur Erfassung und Beschreibung von Populationsdynamik in GP-Prozessen (insbesondere genetische Diversität und Weitergabe von Erbgut) wird erläutert, ebenso wie erweiterte Selektions-Konzepte, zusätzliche Optimierungsstufen, on-line und sliding-window GP-Varianten.

Der zweite Teil der vorliegenden Arbeit faßt die Ergebnisse empirischer Studien zu den im ersten Teil erläuterten GP-Konzepten zusammen. Diese Tests wurden dabei unter Verwendung unterschiedlicher Datensätze ausgeführt: Einerseits wurden Meßdaten von technischen, mechatronischen Systemen verwendet, andererseits

wurden Benchmark-Datensätze aus dem Bereich des maschinellen Lernens verwendet, welche dem UCI machine learning repository entnommen wurden. Die Resultate dieser Testserien demonstrieren nicht nur, daß bzw. wie GP geeignet ist, für verschiedenste Arten von Systemen Modelle zu generieren, sondern bieten auch Einblicke in populationsdynamische Prozesse, welche während der Ausführung von GP-Prozessen auftreten.

Den Abschluß der vorliegenden Arbeit bilden eine Zusammenfassung, eine Bibliographie der referenzierten Arbeiten sowie eine kurze Biographie des Verfassers.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Dissertation selbständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, April 2008 Dipl.-Ing. Stephan M. Winkler

Contents

Chapter 1

Introduction

1.1 Thesis Outline

Except for this short introduction and conclusion and bibliography sections at the end of this thesis, the present work is divided into two major parts: Part I summarizes the most important theoretical concepts that are relevant in the context of evolutionary system identification, and Part II describes and summarizes several empirical test series that have been designed and executed in order to demonstrate not only how, but also how well and why the theoretical concepts described in Part I perform on various kinds of data sets.

Theoretical Aspects

Part I starts with an introduction into evolutionary computation in general in Chapter 2, followed by a detailed summary of theory and practical aspects of genetic programming (GP) in Chapter 3. As we see in this chapter, GP can be seen not only as a powerful extension to the genetic algorithm, but also as the art of evolving computer programs and as a generic concept for the automated programming of computers. The so-called GP cycle [LP02], shown in the left part of Figure 1.1, represents GP's iterative concept of repeatedly creating and testing programs in order to obtain programs that are able to solve the given problem situation.

Chapter 4 describes enhanced selection concepts (for parents as well as offspring selection) which can be used for improving GP's performance in system identification; parallel concepts for genetic algorithms, which can also be used in genetic programming, are summarized in Chapter 5.

A detailed introduction to data based modeling and structure identification is given in Chapter 6: Mathematical expressions modeling the behavior of systems are generated using measurement data; what we want to get in the end is a formula that fits the given data as well as possible (as exemplarily shown in the right part of Figure 1.1). The GP based structure identification approach, that has been implemented using the HeuristicLab framework, is then described in Chapter 7 followed by discussions on regression, time series and classification specific aspects in system identification in Sections 8.1 and 8.2, respectively. Chapter 9 describes various approaches for the incorporation of a priori knowledge into the GP-process, and local model adaptation techniques (pruning and parameter optimization) for GP are summarized in Chapter 10.

The Chapters 11 and 12 are dedicated to GP model similarity measures and population dynamics: The similarity of formulas (in a GP population) is measured with respect to their structure or subtree evaluations, and based on these similarities we can measure the diversity in GP populations. Additionally, concepts for measuring characteristics regarding genetic propagation and variables diversity are also described. Part I on theoretical aspects in GP based system identification is concluded by an overview of approaches for on-line and sliding window GP given in Chapter 13.

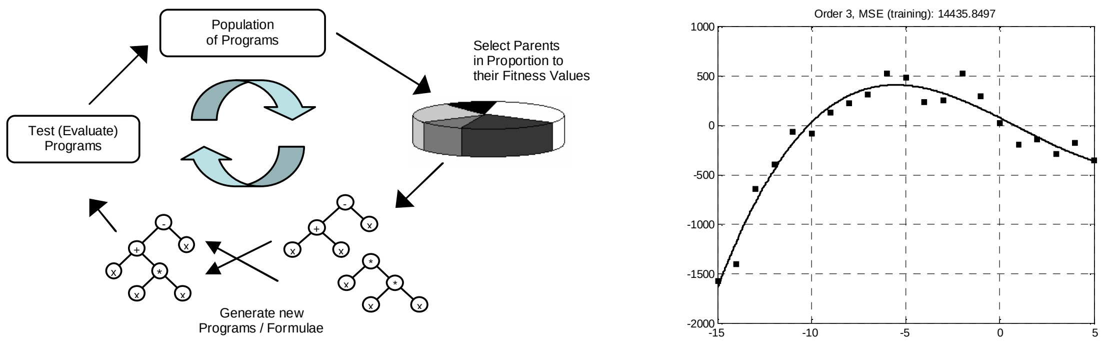

Population of Programs
Select Parents in Proportion to their Fitness Values
Test (Evaluate) Programs
Generate new Programs / Formulae
Order 3, MSE (training): 14435.8497
1000
500
0
-500
-1000
-1500
-2000
-15
-10
-5
0
5

Figure 1.1: Left: The genetic programming cycle [LP02]; right: illustration of a data based modeling example.

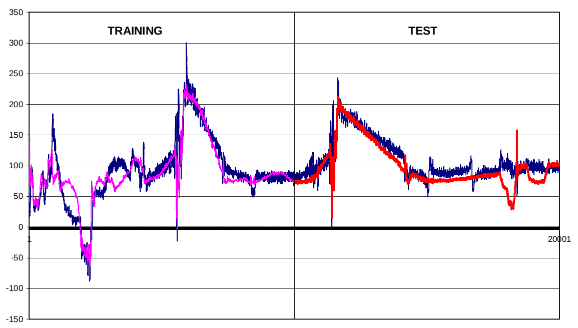
Empirical Studies

As empirical studies with different problem classes and instances are widely considered one of the most effective ways to analyze the potential of heuristic optimization techniques, Part II summarizes several test series that have been executed and analyzed in order to demonstrate how and how well evolutionary system identification works using the concepts described in the first part of this thesis.

The first two chapters of Part II demonstrate the ability of GP to generate appropriate models for time series and classification problems. Chapter 14 summarizes the results obtained for data sets representing measurement data of mechatronical systems; classification models generated using GP for benchmark data sets are reported on in Chapter 15. Figure 1.2 exemplarily illustrates the evaluation of time series and classification models.

Test results in the context of GP in volatile environments, i.e. on-line and sliding window genetic programming, are summarized in Chapter 16. In Chapter 17 the reader can find the analysis of exemplary GP test series regarding genetic propagation and the impact of selection strategies on genetic inheritance distributions, variables diversity, population diversity in single population as well as multi population GP, and code bloat and strategies how to combat it.

We have also tested strategies for incorporating physical knowledge about the formation of NO_x emissions of a diesel engine; test results regarding the generation of virtual sensors for NO_x using a priori knowledge are summarized in Chapter 18. Finally, Part II on the results of practical applications of enhanced GP techniques to system identification is completed by an analysis of the similarity of models produced by GP for different data based modeling tasks in Chapter 19.



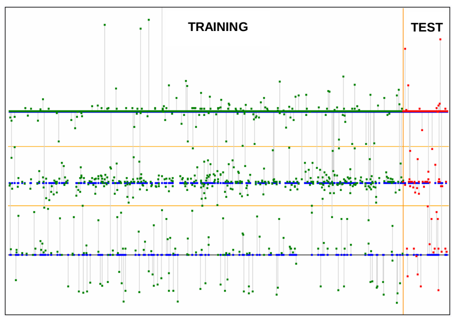


Figure 1.2: Left: Time series model evaluation; right: classification model evaluation including optimal class thresholds.

1.2 Research Project Background

This thesis mainly reflects on research work done within the Translational Research Project L284 "GP-Based Techniques for the Design of Virtual Sensors" sponsored by the Austrian Science Fund (FWF). The following institutions are involved in the execution of this project:

- Department of Software Engineering as well as Research Center Hagenberg, Upper Austrian University of Applied Sciences, Campus Hagenberg,
- Institute for Design and Control of Mechatronical Systems, Johannes Kepler University Linz, and
- Linz Center of Mechatronics (LCM).

The author of this thesis as well as the proposer of the research project mentioned above are members of the Heuristic and Evolutionary Algorithms Laboratory (HEAL) research group, founded by Dr. Michael Affenzeller in 2002 as the Genetic Algorithms Research Group at the former Institute of Systems Theory and Simulation, JKU Linz. The HeuristicLab, a paradigm-independent and extensible environment for heuristic optimization that has been implemented by members of the HEAL research group (mainly by Stefan Wagner), has been used as basic framework for the research work described in this thesis.
Information about the HEAL research group and also the HeuristicLab software framework can be found at the HeuristicLab website[1].

[1]`http://www.heuristiclab.com/`

Part I

Theoretical Aspects

Chapter 2

Evolutionary Computation

2.1 Evolutionary Computation

Basically, there are two main approaches in computer science that copy evolutionary mechanisms: Genetic algorithms (GA) and evolution strategies (ES). Genetic algorithms go back to Holland [Hol75], an American computer scientist and psychologist who developed his theory not only under the aspect of solving optimization problems but also to observe biological processes. Essentially, this is the reason why genetic algorithms are much closer to the biological model than evolution strategies, for which the theoretical foundations were given by Rechenberg and Schwefel ([Rec73], [Sch94]).

In general, evolutionary computation (EC) attempts use populations in which each individual is different from the other ones with respect to their genetic information; the genotype includes parameters which contain all necessary information about the fitness of a certain individual. Before the intrinsic evolutionary process is started, the initial population is initialized (in most cases arbitrarily). Evolution, i.e. replacement of the old generation by a new generation, proceeds until a certain termination criterion is fulfilled.

The major differences between evolution strategies and genetic algorithms lie in the representation of the genotype and the calculation of the fitness of solution candidates as well as in the operators used (mutation, recombination, selection). In contrast to GAs, where the main role of the mutation operator is simply to avoid stagnation, mutation is the primary operator of Evolution Strategies. As a further difference between the two major representatives of evolutionary computation, selection in case of evolution strategies is deterministic which is not the case in the context of genetic algorithms or in nature.

Genetic programming (GP), an extension to genetic algorithms, is a domain-independent, biologically inspired method that is able to create computer programs from a high-level problem statement. In fact, virtually all problems in artificial intelligence, machine learning, adaptive systems, and automated learning can be recast as a search for a computer program; genetic programming provides a way to search for a computer program in the space of computer programs. Similar to GAs, GP works by imitating aspects of natural evolution, but whereas GAs are intended to find an array of characters or integers representing the solution of a given problem, the goal of a GP process is to produce a computer program (or, as in our case, a formula) solving the optimization problem at hand. As in every evolutionary process, repeatedly new individuals (in GP's case, new programs or formulae) are created, tested, and fitter ones of the population succeed in creating children of their own whereas unfit ones are removed from the population.

Figure 2.1 shows a taxonomy of optimization techniques[1]; GAs and their subclass GP belong to the class of evolutionary algorithms and, more general, nature inspired algorithms and stochastic optimization techniques.

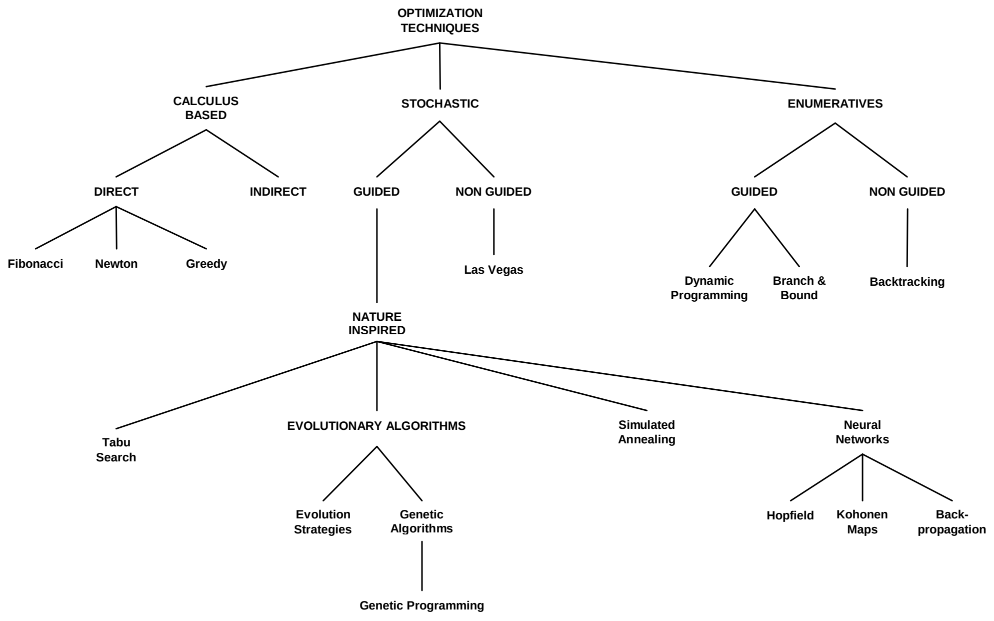


Figure 2.1: Taxonomy of optimization techniques.

[1]This taxonomy is in fact a slightly simplified version of the overview chart given in [AW04].

2.2 Genetic Algorithms

2.2.1 Darwin's Evolution Theory

Charles Robert Darwin, born in 1809 in Shrewsbury, England, was a revolutionary geologist, botanist, naturalist and zoologist who laid the foundation for both the modern theory of evolution and of natural selection as a mechanism. From 1831 to 1836, after graduating from Cambridge with a degree in theology, Darwin joined a British science expedition aboard the H.M.S. Beagle, serving as naturalist. During this worldwide sea voyage, especially on the Galapagos Islands, he studied animals and plants everywhere he went, collected specimen for further studies and so got the inspiration and data for his theories.
Back in England he went on studying the notes and the specimen he had collected during his voyage around the world. Out of his studies he developed several theories; in 1859, he published them in the book "On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life" ([Dar59], revised edition: [Dar98]), which surely remains his most famous work. Darwin continued to write and publish his works on biology, botany, geology and zoology until he died on April 19, 1882; he lies buried in Westminster Abbey.

Some of the most important aspects of Darwin's theories are that evolution surely occurs, that evolutionary changes are gradual, requiring thousands to millions of years, and that the primary mechanism for evolution is a process called natural selection (which he explained especially in Chapter 4 of his book on the origin of species, "Natural Selection; Or the Survival of the Fittest"). Furthermore, Darwin stated that in sexually reproducing species generally no two individuals are identical, and that this variation is heritable. From this it follows that in stable populations, where each individual must struggle to survive, those are more likely to survive that show the "best" characteristics; those advantageous characteristics will be passed on to their offspring and inherited by following generations. As evolution goes on, these desirable characteristics will become dominant among the population: This is what Charles Darwin called *natural selection*.

2.2.2 Basics of Genetic Algorithms

The principles of genetic algorithms (GAs) were first presented by John Holland, an American computer scientist and psychologist, in 1975 [Hol75]. Basically, a GA tries to imitate the biological evolution of a species in order to achieve an (almost) optimal state, that is to produce (almost) optimal solutions for a given problem applying the principles of evolutionary biology to computer science. Detailed overviews of GAs and their implementation in various fields were for example given by Goldberg [Gol89] and Michalewicz [Mic92].

A GA works with a set of candidate solutions (also known as individuals) called population. Each solution candidate is characterized by its list of parameters, in analogy to the terminology of biology also called chromosomes or genome. Originally, chromosomes are represented as simple strings of data and instructions; in general, a chromosome is a n-tuple $(c_1, \ldots, c_n)$ where each position represents one gene storing information for a certain characteristic feature of the represented object. By now a wide variety of other data structures for storing chromosomes has also been used.

When a GA is started, the initial gene pool has to be created (typically at random, or the programmer may "seed" the population intentionally to form an initial pool of possible solutions). This is also called the first generation pool.

During the execution of the algorithm each individual has to be evaluated, which means that a value indicating the "fitness" or "goodness" is returned by a fitness function. This fitness value should quantify the adaptedness of a chromosome in correlation with the algorithm's goal.

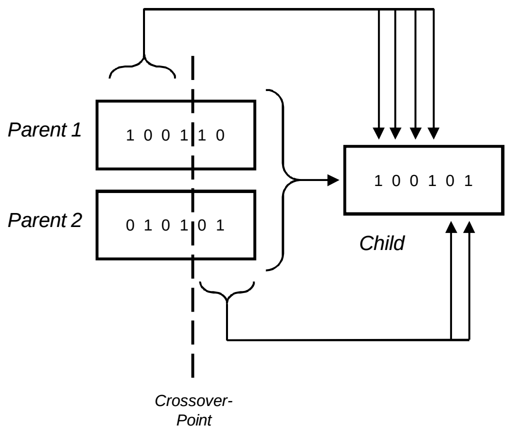
A bit more exactly, the fitness function f is a function $f : \mathcal{S} \rightarrow \mathbb{R}^+$ with $\mathcal{S}$ being the set of all possible solutions to the problem at hand, also called the search space.

After starting the algorithm, new individuals are created in each generation by

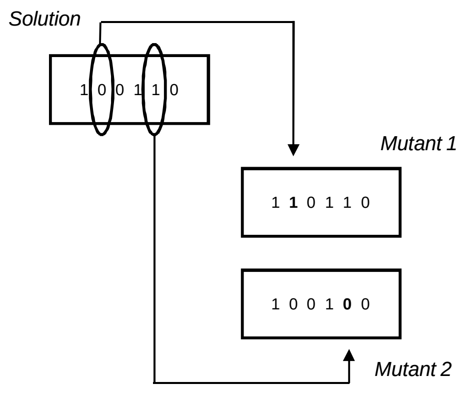
- combining the genetic make-up of two solution candidates (this procedure is called "crossover" or "recombination"), producing a new "child" from two "parents";
- and mutating some individuals, which means that a randomly chosen gene is changed (normally 3-5% of the algorithm's population is mutated in each generation).

In general, crossover is responsible for producing new individuals, whereas mutation is applied to only a small part of the population. Still, the role of mutation shall not be underestimated because it is very important for avoiding so-called premature convergence; without mutation, a genetic algorithm will quite surely get stuck in a local minimum.

Examples of crossover and mutation of solution candidates for a binary encoded problem are shown in Figure 2.2.



(a) Crossover of binary encoded solutions of a GA



(b) Mutations of a binary encoded solution of a GA

Figure 2.2: Exemplary genetic operations in a GA.

Beside crossover and mutation, the third decisive aspect of genetic algorithms is selection, a mechanism in analogy to biology also called “survival of the fittest”. As already mentioned, each individual p_i is associated with a fitness value f_i. Typically, the individual’s probability to inherit its genetic information to the next generation is proportional to its fitness; the better a solution candidate’s fitness value, the higher the probability, that its genetic information will be included in the next generation’s population of the algorithm.

More exactly, there are actually various ways of accomplishing this selection. Those, that are relevant in the context of this thesis (i.e. those, that have been applied during our empirical test series and shall be mentioned in the second part of this thesis), are:

- *Proportional selection*:
  The classical genetic algorithm utilizes this selection method which has been proposed in the context of Holland’s approach. Here the expected number of descendants for an individual i is given by $p_i = f_i/\bar{f}$ with $f : \mathcal{S} \rightarrow \mathbb{R}^+$ denoting the fitness function and $\bar{f}$ representing the average fitness of all individuals ([Hol75], [Aff03]). Therefore, each individual in the population is represented by a space proportional to its fitness. One can imagine this somehow as a roulette wheel, on which every individual gets assigned some space; when the wheel is rotated, that individual, in whose space the ball comes to lie in the

end, is selected.
By repeatedly spinning the wheel, individuals are chosen using random sampling with replacement.

- *Linear-rank selection*:
  In the context of linear-rank selection the individuals in the population are ordered according to their fitness values and copies are assigned in such a way that the best individual receives a predetermined multiple of the number of copies the worst one receives [GB89]. On the one hand, rank selection reduces the dominating effects of very good solution candidates ("super individuals"), but on the other hand also warps the difference between individuals with close fitness values and so increases the selective pressure in stagnative populations. Even although linear-rank selection ignores the information about the relative fitness of different individuals and even violates some of Holland's fundamental ideas[2], it has been used with some success in various situations.

- *Tournament selection*:
  Actually, there is a number of variants of this selection model. The most common one is the so-called k-tournament: Every time an individual has to be selected, k individuals are drawn from the population, and the fittest one of those is presented as the selected individual.

- *Random selection*:
  In random selection individuals are selected from the population completely randomly without regard to their fitness.

This procedure of crossover, mutation and selection is repeated many times (over many generations) until some abort criterion is fulfilled to reach a generation in which all members of the population have good quality values in the sense of the chosen fitness function. This approach is summarized in Algorithm 1 (where a maximum number of generations is used as termination criterion) and graphically shown in Figure 2.3.

One of the main advantages of the genetic algorithm concept is that it combines both directed and undirected search methods: Whereas selection normally picks individuals with respect to their fitness values, crossover and mutation are performed randomly without considering the quality of the involved solution candidates.

[2]One of the basics of Holland's theory of genetic algorithms is his so-called schema theorem; it was described in [Hol75] and is for example summarized and explained in [Aff03]. In the context of this schema theorem, the proportional selection model has been proposed, whereas linear-rank selection violates the schema theorem.

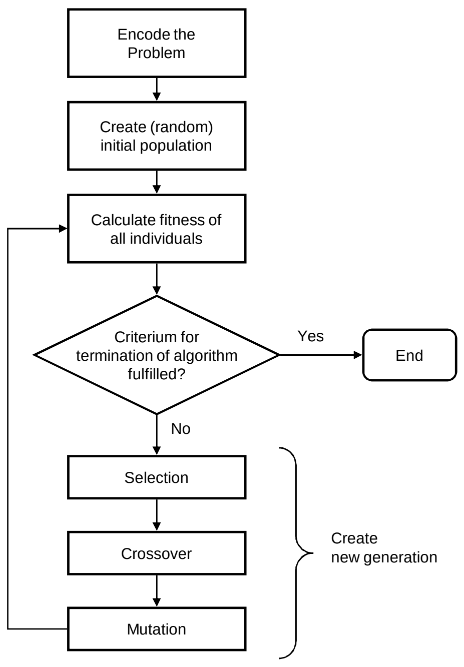

Encode the Problem
Create (random) initial population
Calculate fitness of all individuals
Criterium for termination of algorithm fulfilled?
Yes
End
No
Selection
Crossover
Mutation
Create new generation

Figure 2.3: The standard genetic algorithm (GA).

Algorithm 1 The classical genetic algorithm (GA).

```
Initialize total number of generations $noOfGenerations \in \mathbb{N}$;
Initialize actual number of generations $i = 0$;
Initialize size of population $|POP| \in \mathbb{N}$;
Initialize mutation rate $mutRate \in [0, 1]$;
Initialize elitism rate $elitism \in [0, |POP|]$;
Produce an initial population $POP_0$ with size $|POP|$ randomly

while $i < noOfGenerations$ do
    Initialize next population $POP_{i+1}$

    while $|POP_{i+1}| < (|POP| - elitism)$ do
        Select two individuals from the members of $POP_i$;
        Generate new child by crossing the two selected parents;
        Mutate child with probability $mutRate$;
        Insert child into $POP_{i+1}$
    end while

    Insert $elitism$ best individuals from $POP_i$ into $POP_{i+1}$;
    $i = i + 1$
end while
```

2.2.3 Problem Representation

As already stated before, the first genetic algorithm presented in literature [Hol75] used binary vectors for the representation of solution candidates (chromosomes). Consequently, the first solution manipulation operators (single point crossover, bit mutation) have been developed for binary representation. Furthermore this very simple GA, commonly denoted as the canonical genetic algorithm (CGA), represents the basis for extensive theoretical inspections, resulting in the well known schema theorem and the resulting building block hypothesis ([Hol75], [Gol89]).

This unique selling point of GAs is to compile so-called building blocks, i.e. somehow linked parts of the chromosome which become larger as the algorithm proceeds, advantageously with respect to the given fitness function. In other words, one could define the claim of a GA to assemble the basic modules of highly fit or even global optimal solutions (which the algorithm of course doesn't know at the beginning), which are all already available in the initial population but widespread over many individuals, in such a clever way that continuously growing sequences of somehow linked highly qualified alleles, the so-called building blocks, are formed.

Compared to heuristic optimization techniques based on neighborhood search like tabu search [Glo86] or simulated annealing [KGV83], for example, the methodology of GAs to combine partial solutions (by crossover) is potentially much more robust with respect to getting stuck in local but not global optimal solutions; this tendency of neighborhood based searches denotes a major drawback of these heuristics. Still, when applying GAs the user has to draw much more attention on the problem representation in order to fulfill the claim stated above. In that sense the problem representation must allow the solution manipulation operators, especially crossover; this is because crossover is responsible for combining the properties of two solution candidates which may be located in very different regions of the search space so that valid new solution candidates are built. For that reason the problem representation has to be designed in a way that it allows its certain crossover operators to build valid new children (solution candidates) with a genetic make-up that consists of the allele set union of its parent alleles.

Furthermore, as a tribute to the general functioning of GAs, the crossover operators also have to support the potential development of higher-order building blocks (longer allele sequences). Only if all these solution manipulator properties can be provided by a certain problem representation, the corresponding GA can be expected to function as it should, i.e. in the sense of a generalized interpretation of the building block hypothesis.

2.3 Evolution Strategies

Evolution strategies (ESs), beside GAs the second major representative of evolutionary computation, were developed since the 1960s, primarily by a German research community around Rechenberg and Schwefel at the Technical University of Berlin, and have been extensively studied in Europe (see for example [Rec73], [Sch75] or [Sch94]).

As it is an evolutionary algorithm, the optimization process based on ES is executed by applying operators in a loop, i.e. main operations are applied on the solution candidates repeatedly until a given termination criterion is met. Similar to GAs, an ES works with a population of individuals; each individual is characterized by its parameter vector which is used to calculate the individual's fitness value. In every step of the algorithm's execution (that is, in each generation), the old population is replaced by a new one. Still, there are differences between GAs and ESs, especially in the form of their genotypes, the calculation of the fitness values and the operators (mutation, recombination and selection):

- ESs use real-coding of design parameters, they model the organic evolution at the level of individual's phenotypes; the representation used is a fixed-length real-valued vector. As with the bit- or integer-arrays of genetic algorithms, each position in the vector corresponds to a feature of the individual.
- Whereas GAs use mutation only for avoiding stagnation, mutation is the main reproduction operator in evolution strategies: Each component of the parameter vector is mutated individually in each generation. An additive mutation is carried out, and small mutations are more likely than big ones. The standard mutation distribution is the Gaussian mutation, for it has a lot of advantages: The Gaussian distribution $N(0, \sigma)$ is standardized with average 0 and variance σ^2 (see Figure 2.5). The intensity of the mutation can be adapted by the algorithm during its execution.
- In addition to mutation, recombination can be used to create a new individual (a "child") out of two "parents", too. In general, recombining two ES solution candidates means calculating the geometric average of the parents' parameter vectors.
- In contrast to nature and GAs, the selection of ESs works in a totally deterministic way: In each generation only the best individuals survive, whereas in GAs better individuals (normally) just have higher likelihood to be considered for producing new solution candidates.

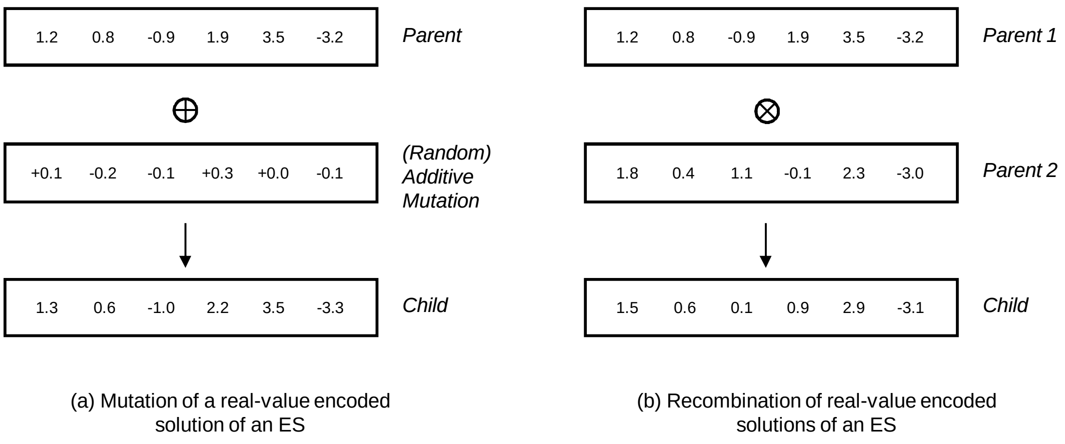


Figure 2.4: Exemplary genetic operations in ES.

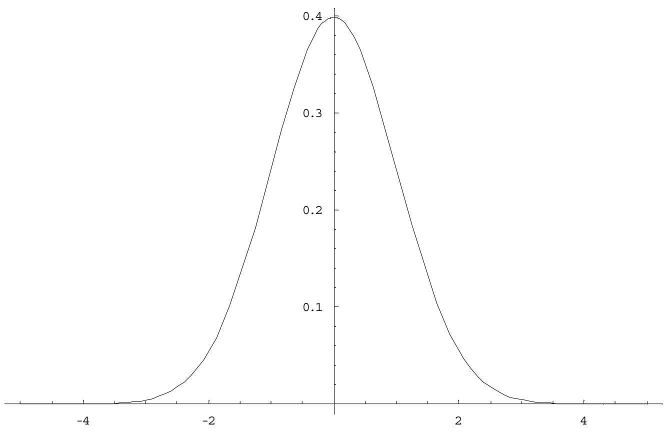


Figure 2.5: The Gaussian probability density function for $N(0,1)$.

Examples for mutation and recombination in the context of ESs are shown in Figure 2.4.

In each generation of an ES algorithm, λ children are produced by μ parent individuals (with $\lambda \gg \mu$); by selection, the best children are chosen and become the parents of the next generation. Typically, parent selection in ES is performed uniformly randomly, with no regard to fitness; survival in ESs simply saves the μ best individuals, which is only based on the relative ordering of fitness values. This form of selection is often referred to as ranking selection, since only the rank of individuals is of importance [SJB+93].

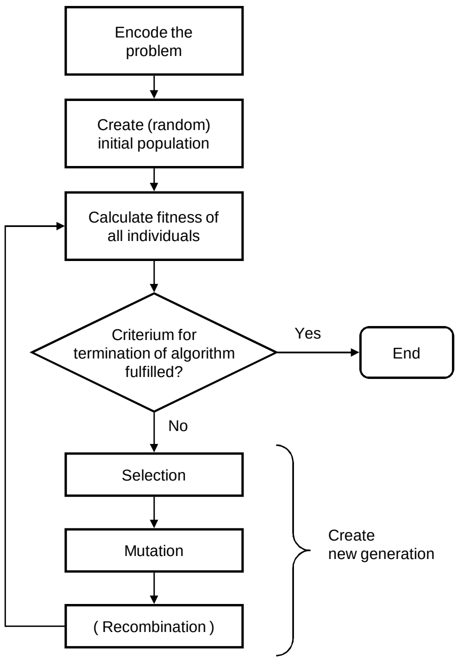

Encode the problem
Create (random) initial population
Calculate fitness of all individuals
Criterium for termination of algorithm fulfilled?
Yes
End
No
Selection
Mutation
(Recombination)
Create new generation

Figure 2.6: The standard evolution strategy (ES) algorithm.

Basically, there are two selection strategies for ESs:

- The (μ, λ)-strategy: μ parents produce λ children; the best μ children are selected and form the next generation's parents. This selection approach is also called the "point-selection".
- The $(\mu + \lambda)$-strategy: If this selection model, also called the "plus-selection", is applied, μ parents produce λ offspring; parents and children form a pool of potential new parents, and the best μ individuals are selected from this pool to become the next generation's parents. Using this strategy has the effect that individuals can survive for indefinitely many generations and that the algorithm is more likely to run into premature convergence.

The main procedure steps of the execution of ES is summarized in Algorithm 2 (where again a maximum number of generations is used as termination criterion) and graphically shown in Figure 2.6.

Rechenberg proposed a heuristic for the adaptation of the mutation variance, the so-called 1/5 success rule[3]:

[3]Actually, this success rule was originally proposed for the special case of a (1+1) ES. Similar

Algorithm 2 The classical evolution strategy (ES) algorithm.

```
Initialize total number of generations $noOfGenerations \in \mathbb{N}$;
Initialize actual number of generations $i = 0$;
Initialize selection operator $SelOp \in \{Plus, Point\}$;
Initialize size of population $\mu \in \mathbb{N}$;
Initialize number of children $\lambda \in \mathbb{N}, \lambda \geq \mu$;
Produce an initial population $POP_0$ with size $\mu$ randomly

while $i < noOfGenerations$ do
  Initialize next population $POP_{i+1}$
  if $SelOp = Point$ then
    Initialize children pool pool with size $\lambda$
  else
    Initialize children pool pool with size $\lambda + \mu$
  end if
  Initialize number of generated children $j = 0$

  while $j \leq \lambda$ do
    Select an individual from the members of $POP_i$;
    Generate new child by mutating the selected parent;
    Insert child into pool;
    $j = j + 1$
  end while

  if $SelOp = Plus$ then
    Insert all individuals from $POP_i$ into pool
  end if
  Insert $\mu$ best individuals from pool into $POP_{i+1}$;
  $i = i + 1$
end while
```

The quotient of the number of the successful mutants (those that improve the population's quality) to all mutants should be about 1/5. If this quotient is greater than 1/5, then the mutation variance should be increased; if the quotient is less than 1/5 (which means that less than 20% of the mutations produce better mutants), the mutation variance should be reduced.
Another possibility of influencing the algorithm with respect to the quotient of successful mutants is adapting μ and λ; for example, increasing λ causes higher selection pressure, whereas decreasing λ or increasing μ reduces it.

rules have also been stated for other ES-variants.

Comparing GAs and ESs, one has to consider the following aspects (as summarized in [Aff01]):
Applied to problems of combinatorial optimization evolution strategies tend to find local optima quite efficiently. But in the case of multimodal test functions, global optima can only be detected by evolution strategies if one of the start values is located in the narrower range of a global optimum. GAs on the other hand are mainly controlled by the crossover operator and therefore, a significant greater part of the search space is taken into account. That is why GAs are usually superior to evolution strategies in finding global optima of multimodal test functions.

Still, ESs show certain advantages in comparison to GAs, too. The selection of parent individuals, for example, is not as constrained as it is in GAs or GP; it is easy to average vectors from many individuals to form an offspring, due to the nature of the real vector representation.

Furthermore, when it comes to designing the encoding of a problem one should consider that in evolution strategies features are considered to be behavioral rather than structural. “Consequently, arbitrary non-linear interactions between features during evaluation are expected which forces a more holistic approach to evolving solutions” [Ang96].

Chapter 3

Evolving Programs: Genetic Programming

In the previous chapter we have summarized and discussed genetic algorithms; it has been illustrated how this kind of algorithms is able to produce high quality results for a variety of problem classes.
Still, a GA is by itself not able to handle one of the most challenging tasks in computer science, namely getting a computer to solve problems without programming it explicitly. As Arthur Samuel stated in 1959 [Sam59], this central task can be formulated in the following way:

How can computers be made to do what needs to be done,
without being told exactly how to do it?

In this chapter we give a compact description and discussion of an extension of the genetic algorithm called **genetic programming (GP)**. Similar to GAs, genetic programming works on populations of solution candidates for a given problem and is based on Darwinian principles of survival of the fittest (selection), recombination (crossover), and mutation; it is a domain-independent, biologically inspired method that is able to create computer programs from a high-level problem statement[1].

Research activities in the field of genetic programming started in the 1980s; still, it took some time until GP was widely received by the computer science community.

[1]Please note that we here define computer programs as entities that receive inputs, perform computations, and produce output.

Since the beginning of the nineteen-nineties GP has been established as a human-competitive problem solving method. The main factors for its widely accepted success in the academic world as well as in industries can be summarized in the following way [Koz92]:

- Virtually all problems in artificial intelligence, machine learning, adaptive systems, and automated learning can be recast as a search for computer programs, and
- genetic programming provides a way to successfully conduct the search in the space of computer programs.

In the following we

- give an overview of the main ideas and foundations of genetic programming in Sections 3.1 and 3.2,
- summarize basic steps of the GP-based problem solving process (Section 3.3),
- report on typical application scenarios (Section 3.4),
- explain theoretical foundations (GP schema theories, Section 3.5),
- discuss current GP challenges and research areas in Section 3.6,
- summarize this chapter on GP in Section 3.7, and finally
- refer to a range of outstanding literature in the field of theory and praxis of GP in Section 3.8.

3.1 Main Ideas and Historical Background

As has already been mentioned, one of the central tasks in artificial intelligence is to make computers do what needs to be done without telling them exactly how to do it. This does not seem to be unnatural since it demands of computers to mimic the human reasoning process - humans are able to learn what needs to be done, and how to do it. In short, interactions of networks of neurons are nowadays believed to be the basis of human brain information processing; several of the earliest approaches in artificial intelligence aimed at imitating this structure using connectionist models and artificial neural networks (ANNs, [MP43]). Suitable network training algorithms enable ANNs to learn and generalize from given training examples; ANNs are in fact a very successful distributed computation paradigm and are frequently used in real-world applications where exact algorithmic approaches are too difficult to implement or even not known at all. Pattern recognition, classification, data based modeling (regression) are some examples of AI areas in which ANNs have been applied in numerous ways. Unlike this network-based approach, genetic algorithms were developed using main principles of natural evolution. As has been explained in Section 2.2, GAs are population-based optimization algorithms that imitate natural evolution: Starting with a primordial ooze of thousands of randomly created solution candidates appropriate to the respective problem, populations of solutions are progressively evolved over many generations using the Darwinian principles.

Similar to the GA, GP is an evolutionary algorithm inspired by biological evolution to find computer programs that perform a user-defined computational task. It is therefore a machine learning technique used to optimize a population of computer programs according to a fitness landscape determined by a program's ability to perform the given task; it is a domain-independent, biologically inspired method that is able to create computer programs from a high-level problem statement (with computer programs being here defined as entities that receive inputs, perform computations, and produce output).

The first research activities in the context of GP have been reported in the early nineteen-eighties. For example, Smith reported on a learning system based on GAs [Smi80], and in [For81] Forsyth presented a computer package producing decision-rules (i.e., small computer programs) in forensic science for the UK police by induction from a database (where these rules are Boolean expressions represented by tree structures). In 1985, Cramer presented a representation for the adaptive generation of simple sequential programs [Cra85]; it is widely accepted that this article on genetic programming is the first paper to describe the tree-like representation and operators for manipulating programs by genetic algorithms.

Even though there was noticeable research activity in the field of GP going on by the middle of the 1980s, still it took some time until GP was widely received by the computer science community. GP is very intensive from a computational point of view and so it was mainly used to solve relatively simple problems until the 1990s. But thanks to the enormous growth in CPU power that has been going on since the 1980s, the field of applications for GP has been extended immensely yielding human competitive results in areas such as data-based modeling, electronic design, game playing, sorting, searching and many more; examples (and respective references) are going to be given in the following sections.

One of the most important GP publications was "Genetic Programming: On the Programming of Computers by Means of Natural Selection" [Koz92] by John R. Koza, professor for computer science and medical informatics at Stanford University who is known as one of the main proponents of the GP idea. Based on extensive theoretical background as well as test results in many different problem domains he demonstrated GP's ability to serve as an automated invention machine producing novel and outstanding results for various kinds of problems. By now there have been three more books on GP by Koza (and his team), but also several other very important publications (for example by Banzhaf, Langdon, Poli and many others); a short summary is given in Section 3.8.

Along with these ad-hoc engineering approaches there was an increasing interest in how and why GP works. Even though GP was applied successfully for solving problems in various areas, the development of a GP theory was considered rather difficult even through the 1990s. Since the early 2000s it has finally been possible to establish a theory of GP showing a rapid development since then. A book that has to be mentioned in this context is clearly "Foundations of Genetic Programming" [LP02] by Langdon and Poli since it presents exact GP schema analysis.

As we have now summarized the historical background of GP, it is now high time to describe how it really works and how typical applications are designed - this is exactly what the reader can find in the following sections.

3.2 Chromosome Representation

As in the context of any genetic algorithm based problem solving process, the representation of problem instances and solution candidates is a key issue also in genetic programming. On the one hand, the representation scheme should enable the algorithm to find suitable solutions for the given problem class, but on the other hand the algorithm should be able to directly manipulate the coded solution representation. The use of fixed-length strings (of bits, characters or integers, e.g.) enables the conventional GA to solve a huge amount of problems and also allows the construction of a solid theoretical foundation, namely the schema theorem. Still, in the context of GP the most natural representation for a solution is a hierarchical computer program of variable size [Koz92].

3.2.1 Hierarchical Labeled Structure Trees

3.2.1.1 Basics

So, how can hierarchical computer programs be represented? The representation that is most common in literature and is used by Koza ([Koz92], [Koz94], [KIAK99], [KKS$^+$03a]), Langdon and Poli ([LP02]), and many other authors is the point-labeled structure tree. Originally, these structure trees were for example seen as graphical representations of so-called S-expressions of the programming language LISP ([McC60], [Que03], [WH87]) which have for example been used by Koza in [Koz92] and [Koz94][2]. We do here not strictly stick to LISP-syntax for the examples given, but the main paradigms of S-expressions are to be used in the following.

The following key facts are relevant in the context of structure-tree based genetic programming:

- All tree nodes are either *functions* or *terminals*.
- *Terminals* are evaluated directly, i.e. their return values can be calculated and returned immediately.
- All *functions* have child nodes which are evaluated before using the children's calculated return values as inputs for the parents' evaluation.
- The probably most convenient string representation is the *prefix* notation, also called *Polish* or *Łukasiewicz*[3] notation: Function nodes are given before the child nodes' representations (optionally using parentheses). Evaluation is executed recursively, depth-first way, starting from the left; operators are thus placed to the left of their operands.
  In case of fixed arities of the functions (i.e., if the numbers of function's inputs is fixed and known), no parentheses or brackets are needed.

[2]In fact, of course any higher programming language is suitable for implementing a GP-framework and for representing hierarchical computer programs. Koza, for example, switched to the C programming language as described in [KIAK99], and the HeuristicLab framework and the GP-implementation, which is realized as plug-ins for it, are programmed in C# using the .NET framework - but this is all explained in further detail in later chapters.

[3]Jan Łukasiewicz (1878–1956), a Polish mathematician, invented the prefix notation which is also the basis of the recursive stack ("last in, first out"; [Ham58], [Ham62]). In reference to his nationality the notation is also referred to "Polish" notation.

In a more formal way this program representation structure schema can be summarized as follows [ES03]:

- Symbolic expressions can be defined using
  - a terminal set T, and
  - a function set F.
- The following general recursive definition is applied:
  - Every $t \in T$ is a correct expression,
  - $f(e_1, \ldots, e_n)$ is a correct expression if $f \in F$, $arity(f) = n$ and $e_1, \ldots, e_n$ are correct expressions, and
  - there are no other forms of correct expressions.
- In general, expressions in GP are not typed (closure property: any $f \in F$ can take any $g \in F$ as argument). Still, as we see in the discussion of genetic operators in Section 3.2.1.3, this might be not true in certain cases depending on the function and terminal sets chosen.

In the following we give exemplary simple programs. We thereby give conventional as well as prefix (not exactly following LISP notation) textual notations:

- (a) `IF (Y>X OR Y<4) THEN i:=(i+1), ELSE i:=0`.
  Prefix notation: `IF(OR(>(Y,X),<(Y,4)),:=(i,+(i,1)),:=(i,0))`.

- (b) $\frac{X+5}{2Y}$. Prefix notation: `DIV(ADD(X,5),MULT(2,Y))`.

Graphical representations of the programs (given as rooted, point-labeled structure trees) are given in Figure 3.1.

3.2.1.2 Evaluation

As already mentioned previously, the execution (evaluation) of GP chromosomes representing hierarchical computer programs as structure trees is done recursively, depth-first way, and starting from the left. In order to demonstrate this we here simulate the evaluation of the example programs given in Section 3.2.1.1; graphical representations are given in Figures 3.2 and 3.3.

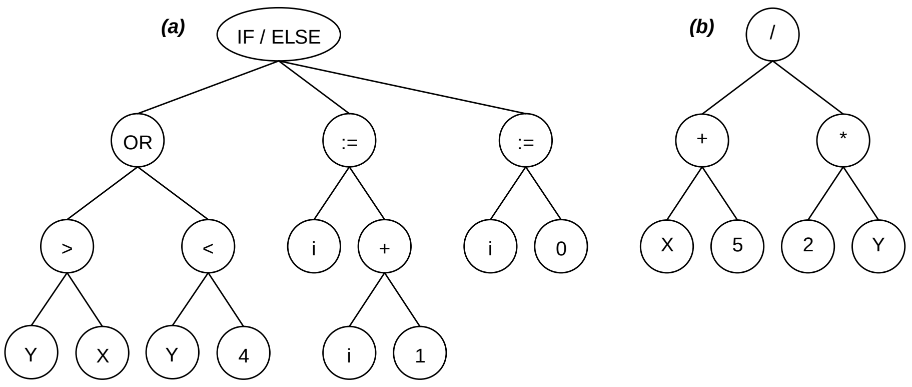


Figure 3.1: Exemplary programs given as rooted, labeled structure trees.

- (a) Internal states before execution: $X = 7$, $Y = 3$, $i = 2$.
  Execution:
  `IF(OR(>(Y,X),<(Y,4)),:=(i,+(i,1)),:=(i,0))`
  $\Rightarrow$ `IF(OR(>(3,7),<(Y,4)),:=(i,+(i,1)),:=(i,0))`
  $\Rightarrow$ `IF(OR(FALSE,<(Y,4)),:=(i,+(i,1)),:=(i,0))`
  $\Rightarrow$ `IF(OR(FALSE,<(3,4)),:=(i,+(i,1)),:=(i,0))`
  $\Rightarrow$ `IF(OR(FALSE,TRUE),:=(i,+(i,1)),:=(i,0))`
  $\Rightarrow$ `IF(TRUE,:=(i,+(i,1)),:=(i,0))`
  $\Rightarrow$ `:=(i,+(i,1))`
  $\Rightarrow$ `:=(i,+(2,1))`
  $\Rightarrow$ `:=(i,3)`.
  Internal states after execution: $X = 7$, $Y = 3$, $i = 3$.

- (b) Internal states before execution: $X = 7$, $Y = 3$.
  Execution:
  `DIV(ADD(X,5),MULT(2,Y))`
  $\Rightarrow$ `DIV(ADD(7,5),MULT(2,Y))`
  $\Rightarrow$ `DIV(12,MULT(2,Y))`
  $\Rightarrow$ `DIV(12,MULT(2,3))`
  $\Rightarrow$ `DIV(12,6)`
  $\Rightarrow$ `2`
  Return value: 2; internal states after execution: $X = 7$, $Y = 3$.

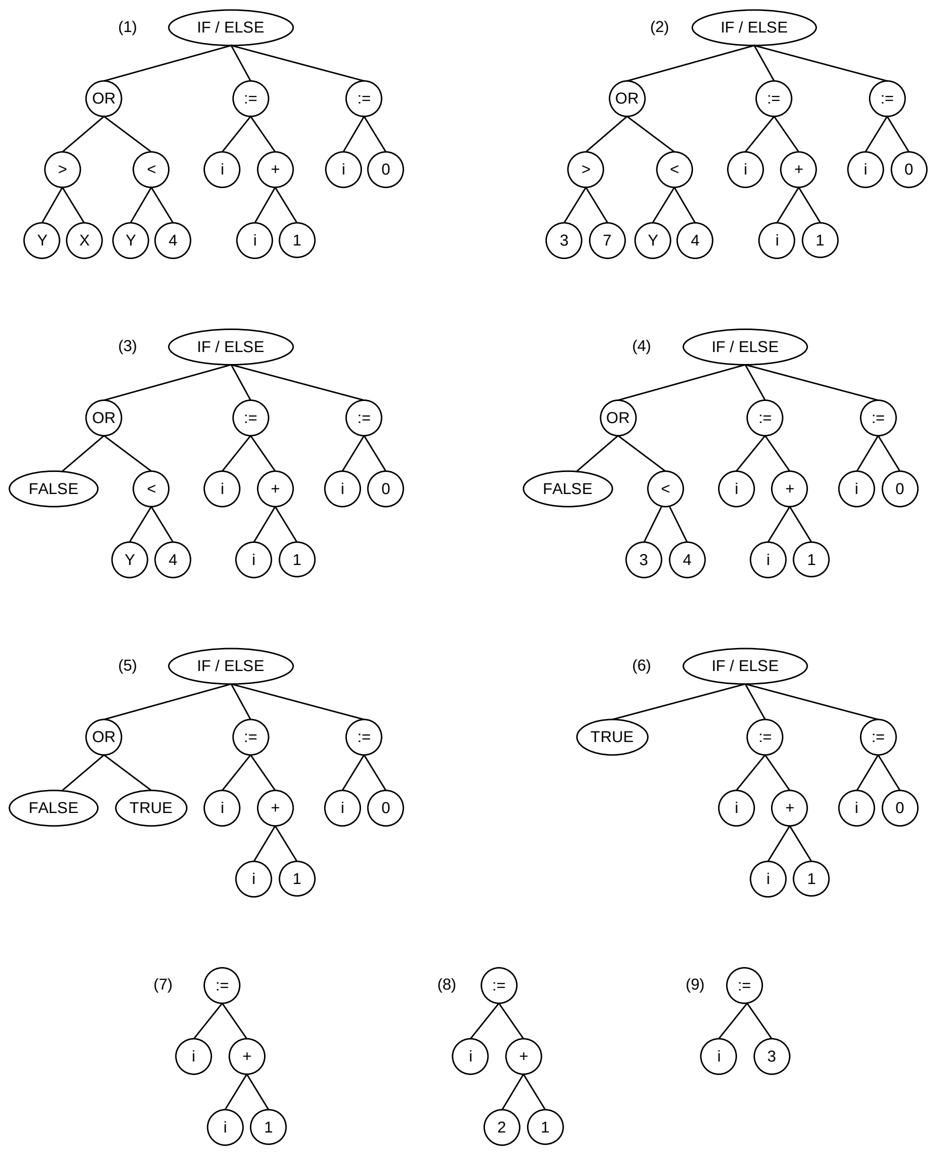


Figure 3.2: Exemplary evaluation of program (a).

3.2.1.3 Genetic Operations: Crossover and Mutation

As genetic programming is an extension to the genetic algorithm, GP also uses two main operators for producing new solution candidates in the search space, namely crossover and mutation.

As we already know from Section 2.2, crossover, the most important reproduction operator, takes two parent individuals and produces new offspring by swapping parts

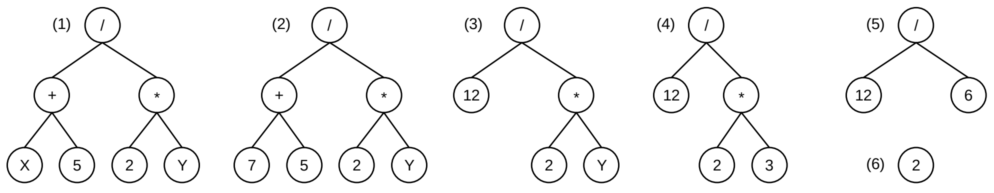


Figure 3.3: Exemplary evaluation of program (b).

of the parents. Here we immediately see one of the major advantages of hierarchical tree representations of computer programs: Single-point crossover can be simply performed by replacing a sub-tree of (a copy of) one of the parents by a sub-tree of the other parent; these sub-trees are chosen at random. There are several different strategies for selecting these sub-trees as it might be reasonable to choose either rather small, rather big, or completely randomly chosen parts.

Mutation can be seen as an arbitrary modification introduced to prevent premature convergence by randomly sampling new points in the search space. In the case of genetic programming, mutation is applied by modifying a randomly chosen node of the respective structure tree:

- A sub-tree could be deleted or replaced by a randomly re-initialized sub-tree.
- A function node could for example change its function type or turn into a terminal node.

Numerous other mutation variants are possible, many of them depending on the problem and chromosome representation chosen. In Chapter 7, for example, we describe mutation variants applicable for GP based structure identification (related to symbolic regression, see Section 3.4.3).

Figure 3.4 illustrates examples for sexual reproduction using the exemplary programs (a) and (b) as parents, labeled as *parent1* and *parent2*, respectively. It thereby becomes obvious that in the context of GP there can be the chance of creating invalid chromosomes: The second offspring (*child2*) seems to be incorrect since it includes the comparison of a Boolean value (`Y>X OR Y<4`) and a number (`2*Y`). Thus, also in GP there are certain constraints that affect the crossover of solution candidates; these constraints have to be considered when it comes to designing and implementing a GP framework.

Of course, it again depends on the chosen implementation if the evaluation of this syntactically dubious program can be executed or not. In case of real-valued representation of Boolean values (`TRUE` represented by 1.0, `FALSE` represented by 0.0, e.g.) this structure tree represents a valid program that can be calculated without any further problems.

Figure 3.5 illustrates exemplary results of applying mutation to program (a). In the first case, a Boolean function node ($<$) is turned into another type of Boolean function node ($>$) yielding *mutant1*; *mutant2* is produced by omitting a sub-tree, namely the second child of the OR function node. While these two first mutants are syntactically correct, *mutant3* is an example for an invalid mutation example: The first child of the conditional (IF) node has been deleted leaving the root node with only two children - the evaluation of this program is not possible.
Again, real-valued representation of Boolean values can help here. In this case the value calculated by the first child of such a conditional node would have to be interpreted as a Boolean value triggering the execution of the second child sub-tree, the *then*-branch. As there is no third child node there is also no *else*-branch, thus there is probably no action if the first (condition) node is evaluated (or at least interpreted) as *false*.

These two examples of syntactically incorrect programs demonstrate what was hinted in Section 3.2.1.1: Even though expressions are in general not typed in GP, there are cases in which this is not true - a fact which has to be considered during the design and implementation of a GP based problem solving system.

3.2.1.4 Advantages

As we are going to see later, the hierarchical structure tree is not the only way how programs can be modeled and used in the GP process. Still, the cumulation of the following reasons strongly favors the choice of this program representation schema[4]:

- Even though structure trees show an (at least for many people) rather unusual appearance and syntax, most programming language compilers internally convert given programs into parse trees representing the underlying programs (i.e., their compositions of functions and terminals). In most programming languages, these parse trees are not (conveniently) accessible to the programmer; here we present the programs directly as parse trees as we need to genetically manipulate parts of the programs (sub-trees).
- As evaluation is executed recursively starting from the root node, a newly

[4]In fact, these reasons partially correlate to Koza's reasons for choosing LISP for his GP implementation reported on in [Koz92] and [Koz94], e.g.

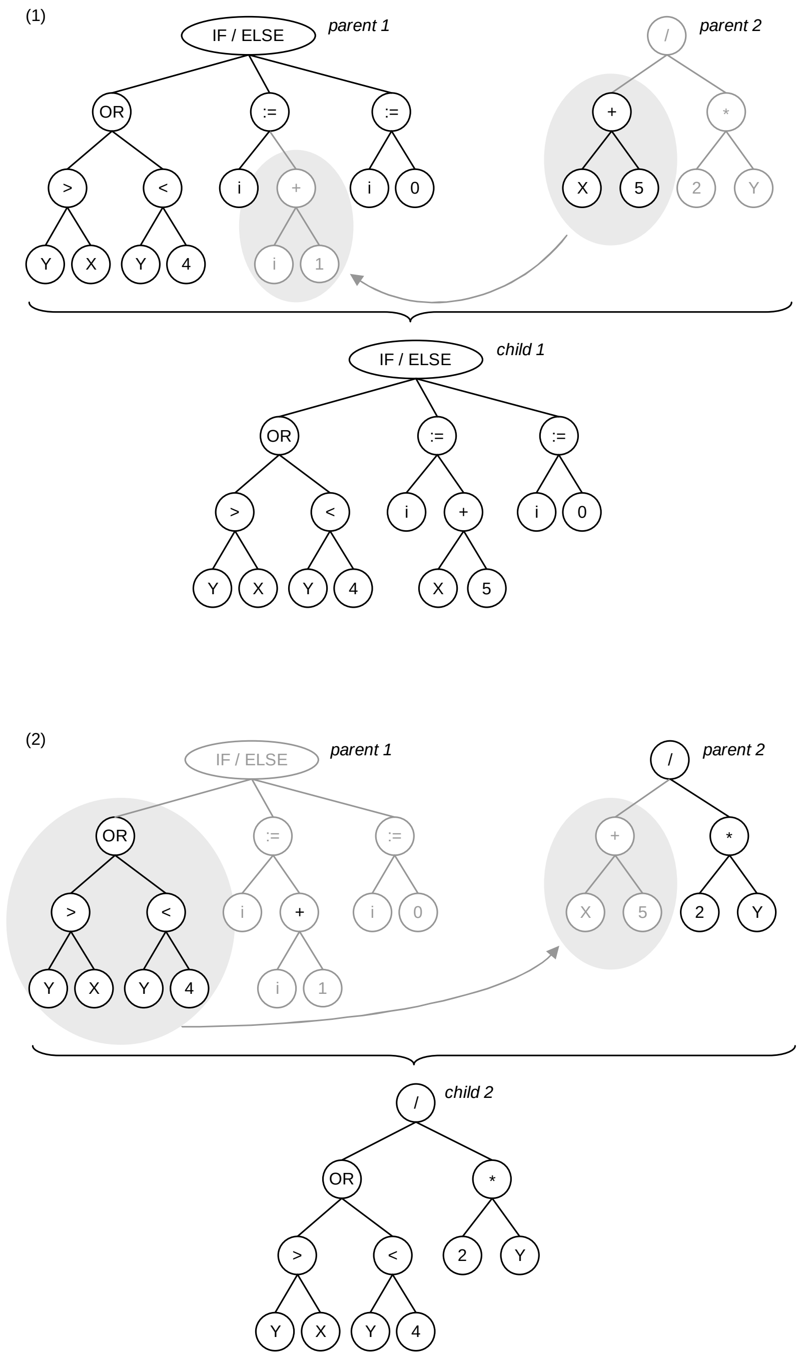


Figure 3.4: Exemplary crossover of programs (a) and (b) labeled as *parent1* and *parent2*, respectively. *Child1* and *child2* are possible new offspring programs formed out of the genetic material of their parents.

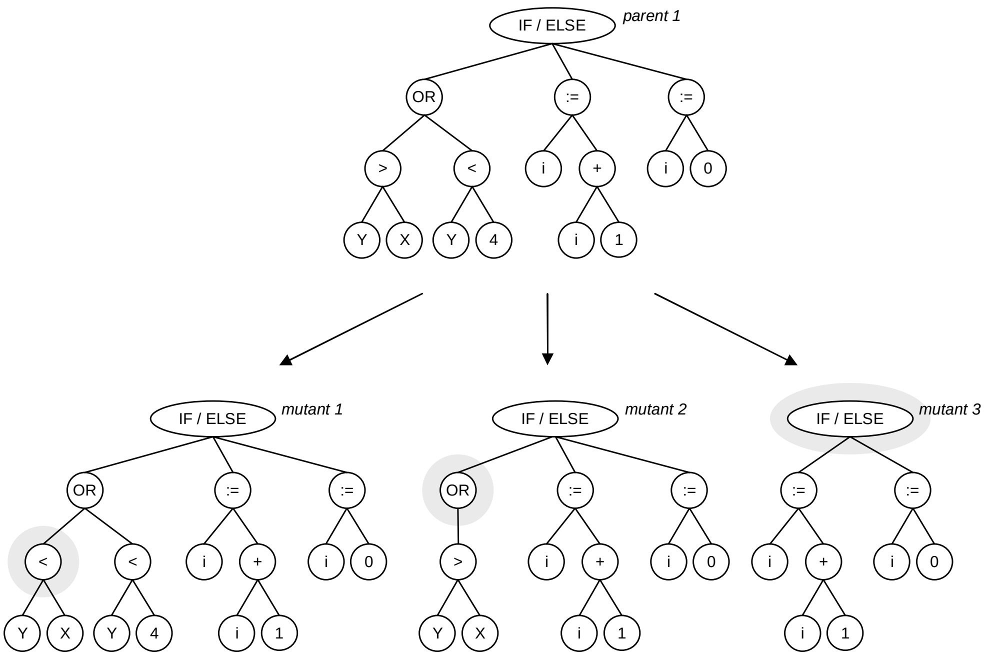


Figure 3.5: Exemplary mutation of a program: The programs *mutant1*, *mutant2*, and *mutant3* are possible mutants of *parent*.

generated or manipulated program can be (re-)evaluated immediately without any intermediate transformation step.

- Structure trees allow the representation of programs whose size and shape change dynamically.

3.2.2 Modular Genetic Programming

Numerous variations and extensions to Koza's structure tree based genetic programming have been proposed since its publication at the beginning of the 1990s. The probably best known and most frequently used one is the concept of *automatically defined functions* (ADFs) proposed in "Genetic Programming II: Automatic Discovery of Reusable Programs" [Koz94].

The main idea of ADFs is that program code (which has been evolved during the GP process) is organized into useful groups (subroutines); this enables the parameterized reuse and hierarchical invocation of evolved code as functions that have not been taken from the original functions set F but are rather defined automatically.

The (re-)use of subroutines (subprograms, procedures) is enabled in this way. In the meantime the idea of ADFs has been extended; automatically defined iterations, loops, macros, recursions and stores have since then been proposed and their use demonstrated for example in [Koz94], [KIAK99], and [KKS$^+$03a].
With ADFs a GP chromosome program is split into a main program tree (which is called and executed from outside) and arbitrarily many separate trees representing ADFs. These separate functions can take arguments as well as be called by the main program or another ADF.

Different approaches realizing modular genetic programming which have gained popularity and are well known in the GP community are the Genetic Library (presented by Angeline in [Ang93] and [Ang94], e.g.) and the Adaptive Representation through Learning (ARL) algorithm (proposed by Rosca, see for example [Ros95a] or [RB96]). In both approaches, some parts of the evolved code are automatically extracted from programs (usually of those that show rather good fitness values). These extracted code fragments are then fixed and kept in the GP library, thus they are available for the evolving programs in the GP population.

Other advanced GP concepts that extend the tree concept are discussed in [Gru94], [KBAK99], [Jac99], and [WC99]; basic features of these modular GP approaches can be combined with multi-tree (multi-agent) systems which shall be described a bit later.

3.2.3 Other Representations

We are not going to say much about not tree based GP systems in the context of this thesis; still, the reader could be prone to suspect that there might be computer program representations other than the tree-based approach. In fact, there are two other forms of GP that shall be mentioned here whose program encoding differs significantly from the approach described before: *Linear* and *graphical genetic programming.*

3.2.3.1 Linear Genetic Programming

The main difference between linear GP and tree based GP is that in linear GP individuals of the GP algorithm (the programs) are not represented by structure trees but rather by linear chromosomes. These linear solutions represent lists of computer instructions which are executed linearly.

Linear GP chromosomes are more similar to those of conventional GAs; however, their size is usually not fixed so that a GP population is likely to contain chromosomes of different sizes which is usually not the case with conventional GA approaches. On the one hand this of course brings along the loss of the advantages mentioned in Section 3.2.1.4, but on the other hand this schema easily enables the representation of *stack based* programs, *register based* programs, and *machine code.*

- In general, a stack is a data structure based on the "last in first out" principle. If a program instruction is to be evaluated, it takes (pops) its arguments from the stack, performs the calculation, and writes back the result by adding (pushing) it back to the top of the stack. A chromosome in *stack-based GP* represents exactly such a stack based program by storing the program instructions in a list and using a stack for executing the program. A typical example can be seen in Perkins' article "Stack-Based Genetic Programming" [Per94], a recent implementation has for example been presented in [HRv07].
- *Register based* and *machine code* GP are essentially similar [LP02]: In both cases data is stored in (a rather small number of) registers, and instructions read data from and write results back to these registers. Initially, a program's inputs are written to registers, and after executing the program the results are given in one or more registers. The main difference between these two GP approaches is the following:
  - Programs in register based GP (as also those of any other kind of GP system) have to be interpreted, i.e. they are executed indirectly or compiled before execution.
  - On the contrary, programs in machine code GP consist of real hardware machine instructions; thus, these programs can be executed directly on a computer. The execution of machine code GP programs is therefore a lot faster than the evaluation of programs in traditional implementations. Nordin's Compiling Genetic Programming System (CGPS) [Nor97] for example presents an implementation of machine code GP.

3.2.3.2 Graphical Genetic Programming

Parallel Distributed Graphical Programming (PDGP, [Pol97], [Pol99b]) is a form of GP in which programs are represented as graphs representing functions and terminals as nodes; links between those nodes define the flow of control and results. PDGP defines a fixed layout for the nodes whereas the connections between them

and the referenced functions are evolved by the GP process. PDGP enables a high degree of parallelism as well as an efficient and effective reuse of partial results; furthermore, it has been shown that it performs better than conventional tree based GP on a number of benchmark problems.
Figure 3.6 shows the graphical, intron-augmented representation of an exemplary program in PDGP (adapted from [Pol99b]).

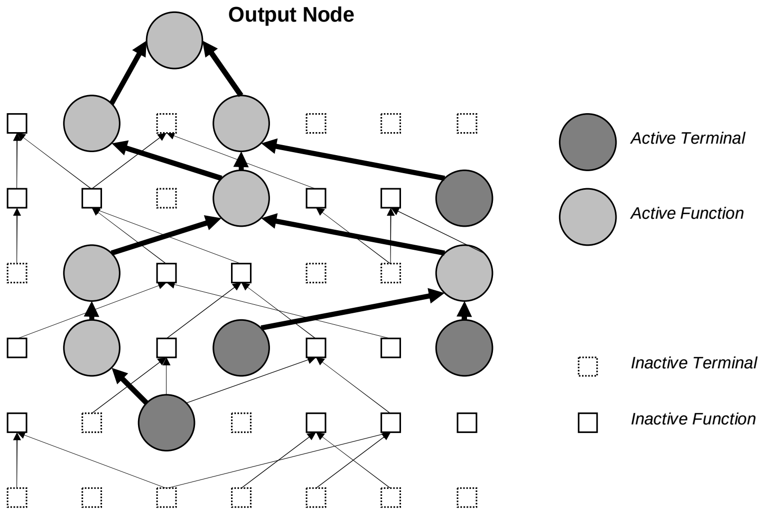


Figure 3.6: Intron-augmented representation of an exemplary program in PDGP [Pol99b].

3.3 Basic Steps of the GP Based Problem Solving Process

3.3.1 Preparatory Steps

Before the GP process can be started there are several preparatory steps that have to be executed. As explained in Section 3.2.1.1, the *function* and *terminal* sets (F and T, respectively) have to be determined. Furthermore, as in any GA application, a *fitness measurement function* also has to be established so that a solution candidate can be evaluated and its fitness can be measured (either explicitly or implicitly).

In addition to these preparations that directly affect the construction and management of individuals of the GP population, there are also some things to be done regarding the execution of the GP algorithm:

- Parameters that control the GP run have to be set,
- a termination criterion has to be defined, and
- a result designation method has to be defined (as explained later in Section 3.3.4).

These preparations in fact have to be done for any genetic algorithm; a similar summary is for example given in [KIAK99]. Figure 3.7 summarizes the major preparatory steps for the basic GP process.

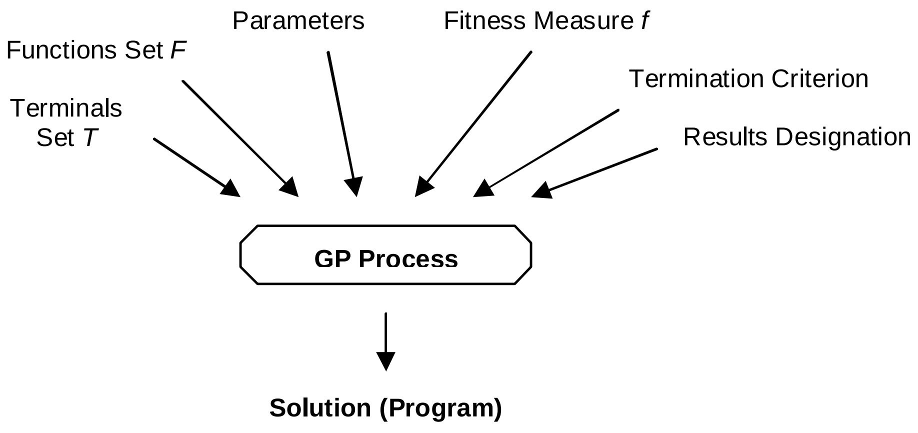


Figure 3.7: Major preparatory steps of the basic GP process.

3.3.2 Initialization

At the beginning of each GA and GP execution the population is initialized arbitrarily before the intrinsic evolutionary process can be started. This initialization can be done either completely at random or using certain (problem specific) heuristics.

For hierarchical program structures as used in GP the random initialization utilizes a maximum initial tree depth D_{max}. As introduced in [Koz92] and for example reflected on in [ES03], there are two possibilities for creating random initial programs:

- Full method: Nodes at depth $d < D_{max}$ point to randomly chosen functions from function set F, and nodes at depth $d = D_{max}$ are randomly chosen terminals (from terminal set T);

- Grow method: Nodes at depth $d < D_{max}$ become either a function or a terminal (randomly chosen from $F \cup T$), and nodes at depth $d = D_{max}$ are again randomly chosen terminals (from T).

The so-called ramped half-half GP initialization method, proposed by Koza [Koz92], has meanwhile become one of the most frequently used GP initialization approaches [ES03]. Both methods, grow and full, are hereby applied, each delivering parts of the initial population.
Still, there is research work going on regarding this issue of finding optimal initialization techniques as it is a fact that the use of different initialization strategies can lead to very different overall results (as for example demonstrated in [HHM04]). For example, there are approaches that produce initial populations that are generated adequately distributed in terms of tree size and distribution within the search space [GAMRRP07].
Later (in Chapter 7) we describe how the definition of structural limits for initial programs has been implemented not via their tree depth but rather via their tree size (i.e., the number of nodes involved) S_{max}.

3.3.3 The Genetic Process: Breeding Populations of Programs

After preparing the GP process and initializing the population, the genetic process can be started. As it is the case in any GA, new individuals (programs) are created using recombination and mutation, tested, and become a part of the new population. Fitter individuals have a bigger chance to succeed in creating children of their own; thus, optimization happens during the run of the evolutionary algorithm. Unfit programs (and with them also their genetic material) wither out of the population. As populations cannot grow infinitely in most applications, new programs somehow have to replace old ones that die off. There are in fact several ways how this replacement can be done:

- Generational replacement: The entire population is replaced by its descendants. This corresponds to generations changes in nature when for example annual plants or animals die in winter whereas their eggs (hopefully) survive; thus, the next generation of the species is founded.

- Steady state replacement: New individuals are produced continuously, and the removal of old individuals also happens continuously. Analogies in nature are obvious as this is more or less how for example human evolution happens.

- Selection of replaced programs: The individuals removed can be either chosen from the unfit ones (worst replacement), from the older ones (replacement with aging), or at random (random replacement), e.g.

This whole procedure is graphically displayed in Figure 3.8 (adopted from [LP02]).

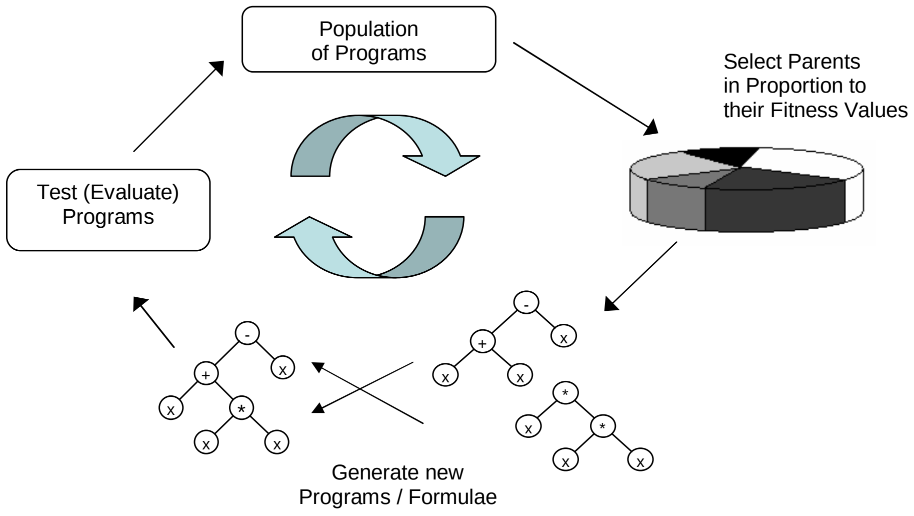


Figure 3.8: The genetic programming cycle [LP02].

In fact, the whole genetic programming process involves more than what is displayed in Figure 3.8: The preparatory steps summarized in Section 3.3.1 also have to be considered, and of course a validation of the results produced has to be done that might lead to a re-formulation of the pre-conditions. A more comprehensive overview of the GP process is given in Figure 3.9.

The execution of the GP cycle is – as GP is an extension to the GA – similar to the cyclic execution of the GA: Solutions are selected from the population, by crossing them they become parents, mutation is applied with a rather small probability, and thus new offspring is produced. In the generational replacement scheme this is repeated until the next generation's population is complete; in the steady state scheme there is no generational cycle but this procedure is also repeated over and over again. The whole procedure is repeated until some pre-defined termination criterion is met (see Section 3.3.4 for details).

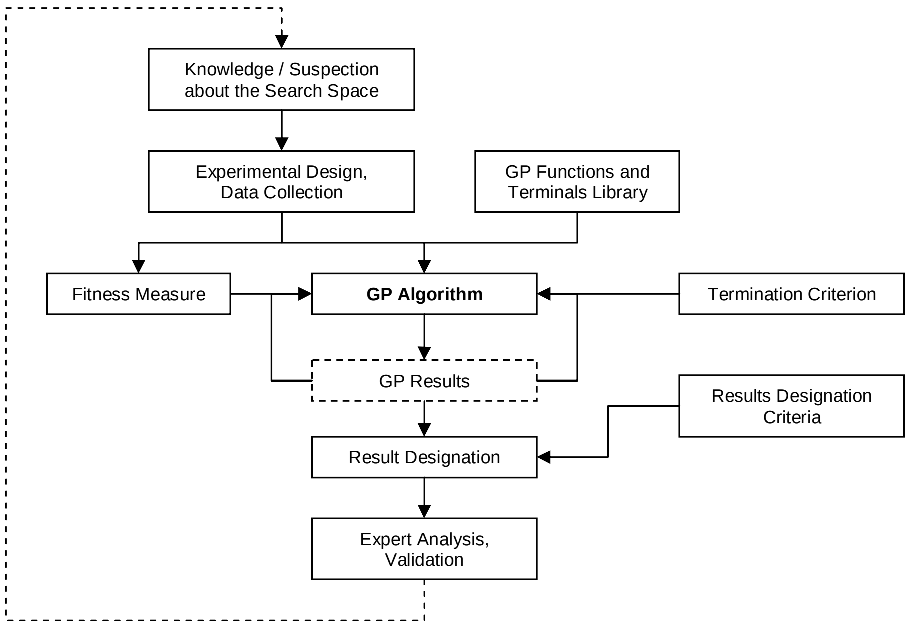

Knowledge / Suspection about the Search Space
Experimental Design, Data Collection
GP Functions and Terminals Library
Fitness Measure
GP Algorithm
Termination Criterion
GP Results
Results Designation Criteria
Result Designation
Expert Analysis, Validation

Figure 3.9: The GP based problem solving process.

In fact, there is a veritable difference in the descriptions of this cyclic workflow for GAs and for GP regarding the offspring creation scheme applied[5]:

- In GAs, crossover and mutation are used sequentially, i.e. both are applied (with mutation having a rather small probability).
- In GP, crossover or mutation (or a simple copy action) are executed independently; each time a new offspring is to be created, one of these variants is chosen probabilistically.

In fact, some researchers even recommend the GP-like offspring creation schema for all evolutionary computation systems (as for example given by Eick, see [Eic07]).

3.3.4 Process Termination and Results Designation

In general, the termination criteria of genetic algorithms are also applicable for genetic programming. A termination criterion might monitor the number of generations and terminate the algorithm as soon as a given limit is reached. Problem

[5]The GA workflow was described in detail in Section 2.2; the GP workflow as it is summarized here is also described in further detail in [Koz92], [KKS+03a], and [ES03], e.g.

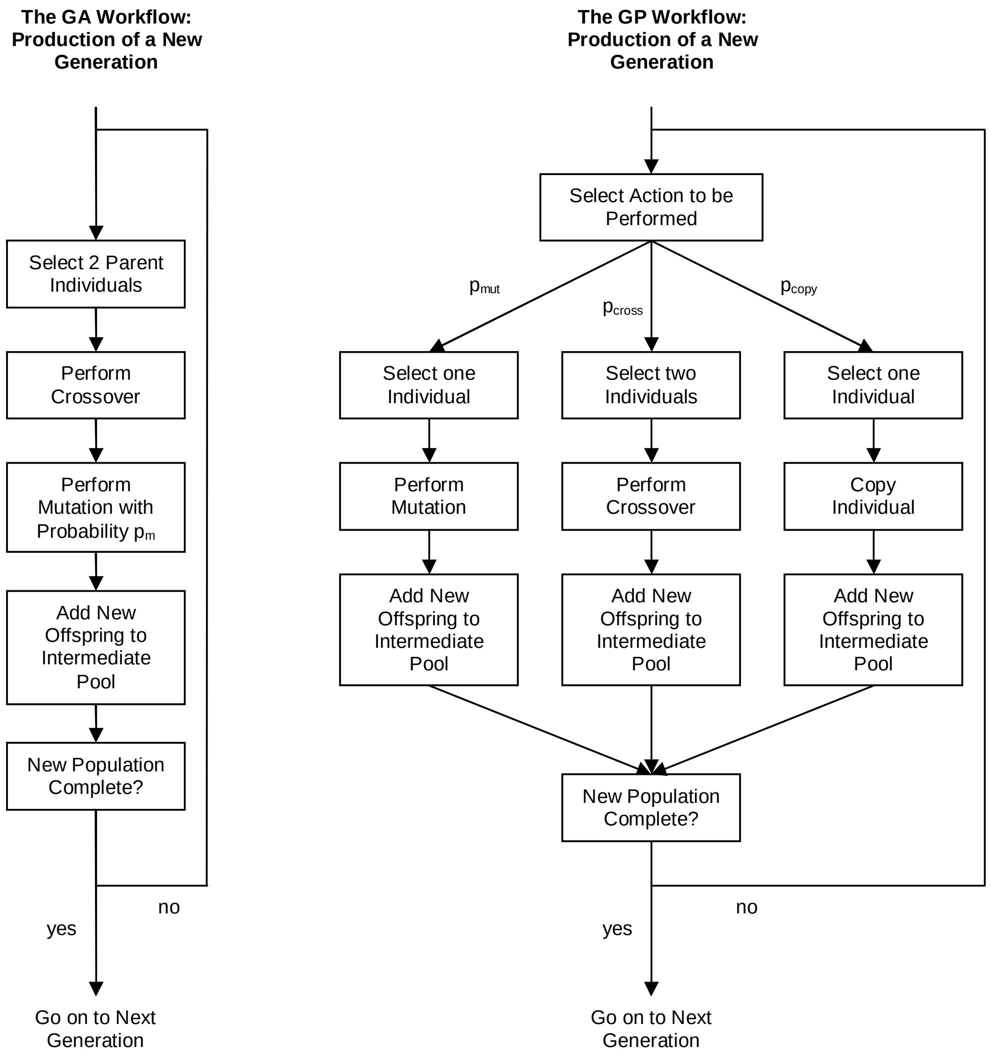

The GA Workflow: Production of a New Generation
Select 2 Parent Individuals
Perform Crossover
Perform Mutation with Probability p_m
Add New Offspring to Intermediate Pool
New Population Complete?
no
yes
Go on to Next Generation
The GP Workflow: Production of a New Generation
Select Action to be Performed
p_mut
p_cross
p_copy
Select one Individual
Select two Individuals
Select one Individual
Perform Mutation
Perform Crossover
Copy Individual
Add New Offspring to Intermediate Pool
Add New Offspring to Intermediate Pool
Add New Offspring to Intermediate Pool
New Population Complete?
no
yes
Go on to Next Generation

Figure 3.10: GA and GP flowcharts: The conventional genetic algorithm and genetic programming.

specific criteria are also used frequently, i.e. the algorithm is terminated as soon as a problem-specific success predicate is fulfilled. In practice, one may manually monitor and manually terminate the run when the values of fitness for numerous successive best-of-generation individuals appear to have reached a plateau [KKS$^+$03a].

After terminating the algorithm it comes to the designation of the result returned by the algorithm. Normally, the single best-so-far individual is then harvested and designated as the result of the run [KKS$^+$03a]. As we show in Chapter 7 there are applications (as for example data based structure identification) in which this is not the optimal strategy. In this case the use of a validation data set V is suggested, i.e. a data collection that was not used during the GP training phase; we eventually test the programs on V and pick the one that performs best on V.

3.4 Typical Applications of Genetic Programming

As genetic programming is a domain-independent method, there is an enormous number of applications for which it has been used for automatically producing solutions of high quality. Here we give a very short summary of exemplary problem classes which have been used for demonstrating GP's power in automatically learning programs for solving problems for more than 15 years, namely the automated learning of multiplexer functions (Section 3.4.1), the artificial ant (3.4.2), and symbolic regression (3.4.3). Finally, in Section 3.4.4 we give a short list of various problems for which GP has proven to be able to produce high quality results.

3.4.1 Automated Learning of Multiplexer Functions

The automated learning of functions requires the development of compositions of functions that can return correct values of functions after seeing only a relatively small number of specific examples; these training samples are combinations of values of the function associated with particular combinations of arguments.

The problem of learning Boolean multiplexer functions has become famous as a benchmark application for genetic programming since Koza's work on it for example presented in [Koz89] and [Koz92]. The input to a Boolean k-multiplexer function is a bit-string consisting of k address bits a_i and 2^k data bits d_i; normally, the bits are thereby aligned following the form $[a_{k-1} \dots a_1 a_0 d_{2^k-1} \dots d_1 d_0]$. The value returned by the multiplexer function is the value of the particular data bit that is addressed by the k address bits. For example, let k be 3 and the three address bits $a_2 a_1 a_0 = 101$, then the multiplexer singles out data bit d_5 to be its output[6]. The abstract black box model of the Boolean multiplexer with three address bits and $2^3 = 8$ data bits as well as the concrete addressing of data bit d_5 is displayed in Figure 3.11.

A solution to this problem obviously has to be a function that uses input in-

[6]Data bit d_5 is in fact the sixth data bit since if $a_2 a_1 a_0 = 000$ data bit d_0 is addressed, so the indices of these data bits are zero-based.

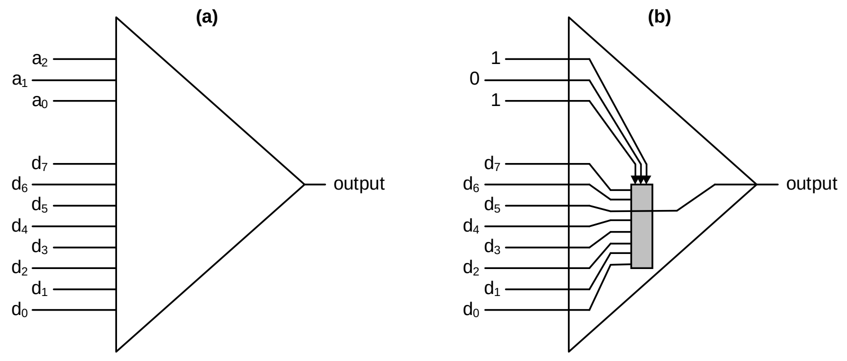


Figure 3.11: The Boolean multiplexer with three address bits; (a) general black box model, (b) addressing data bit d_5.

formation a and d and calculates a Boolean return value. Thus, the terminal has $(k + 2^k)$ elements which correspond to the inputs to the multiplexer; in the case of $k = 3$ the terminal set $T = \{A_0, A_1, A_2, D_0, D_1, \ldots, D_7\}$. The functions used contain Boolean functions and the conditional function, i.e. $F = \{AND, OR, NOT, IF\}$. The evaluation of a solution candidate is done applying the formula to all possible input bit combinations and counting the number of correct output values. As there are $(k + 2^k)$ inputs to the Boolean multiplexer, the number of possible input combinations is (2^{k+2^k}); in the case of $k = 3$, the number of possible input combinations is 2048.
Koza was able to show that GP is able to solve the 3-address multiplexer problem 100% correctly [Koz92]; this optimal result is shown in Figure 3.12. Of course, various test series have been documented in which GP was used for solving problem with multiplexers with more address bits in numerous publications.

3.4.2 The Artificial Ant

The artificial ant problem ([CJ91a], [CJ91b], [JCC+92]) has also been a frequently used benchmark problem for GP since Koza's application [Koz92]; meanwhile, it has become a well-studied problem in the GP community (see for example [LW95], [IIS98], [Kus98], [LP98], and [LP02]).

In short, the problem is to navigate an artificial ant on grid consisting of 32 $\times$ 32 cells. The grid is toroidal so that if the ant moves off the edge of the grid, it reappears and continues on the opposite edge. On this grid, "food" units are distributed (normally along a trail); each time the ant enters a square containing

```
(IF A0 (IF A2 (IF A1 D7 (IF A0 D5 D0))
              (IF A0 (IF A1 (IF A2 D7 D3) D1) D0))
       (IF A2 (IF A1 D6 D4)
              (IF A2 D4 (IF A1 D2 (IF A2 D7 D0)))))
```

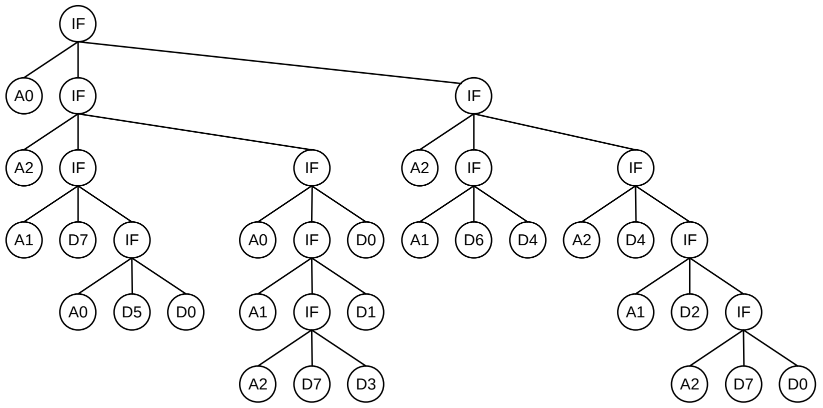

IF
A0 IF IF
A2 IF IF A2 IF IF
A1 D7 IF A0 IF D0 A1 D6 D4 A2 D4 IF
A0 D5 D0 A1 IF D1 A1 D2 IF
A2 D7 D3 A2 D7 D0

Figure 3.12: 100% Correct solution to the 3-address Boolean multiplexer problem [Koz92].

food, the ant eats it. At the beginning of the ant's wanderings it starts at cell $(0, 0)$ facing in a particular direction (east, e.g.); at each time step, the ant is able to move forward in the direction it is facing, to turn right, or to turn left. The goal is to find a program that is able to navigate the ant so that as many food items as possible are eaten in a certain number of time units. The program can use the following:

- Three operations are available, namely `Move`, `Left`, and `Right` which let the ant move ahead, turn left or turn right, respectively; these operations are used as terminals in the GP process.
- The sensing function `IfFoodAhead` investigates the cell the ant is currently facing and then executes the first child operation if food is ahead or the second child action otherwise.
- Additionally, two more functions are available: `Prog2` and `Prog3` take two and three arguments (operations), respectively, which are executed consecutively.

The most frequently used trail is the so-called "Santa Fe trail" designed by Christopher Langton. This trail is displayed in Figure 3.13 (copied from [LP02]); the ant is allowed to wander around the map for 600 time units. This problem is in fact considered a hard problem for GP; thorough explanations for this statements are for example given by Langdon and Poli in "Why ants are hard" ([LP98] and [LP02]). What makes it so hard is not that it is difficult to find correct solutions but rather

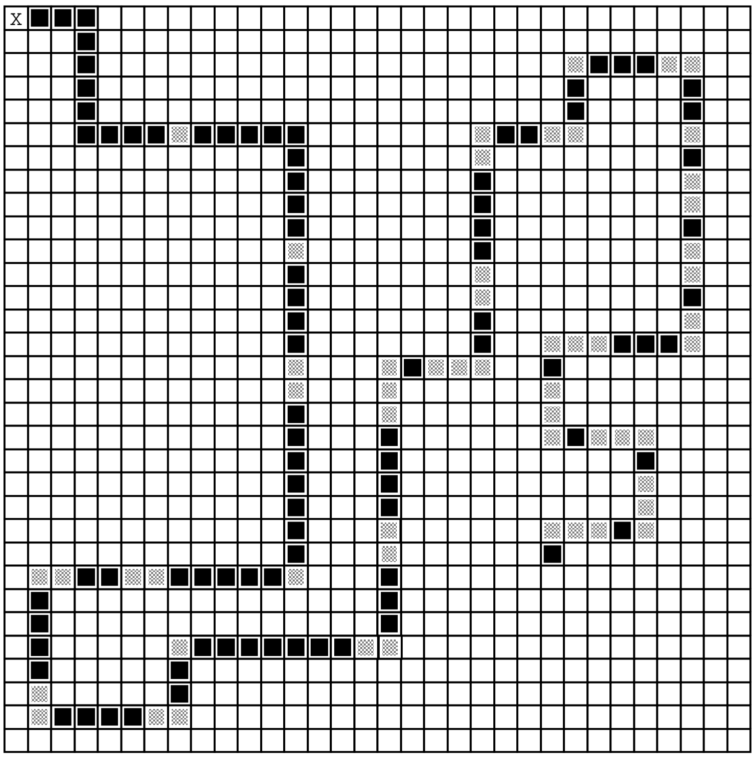


Figure 3.13: The Santa Fe trail.

to find these efficiently and significantly better than random search. As is listed in [LP98], the smallest solutions that solve the Santa Fe trail problem (i.e., those that provide programs that let the ant eat all food packets) are of length 11[7]; one of them is exemplarily shown in Figure 3.14.

Even though it is a very "simple" problem, the artificial ant problem still provides a good basis for many theoretical investigations in GP such as building blocks and schema analysis [LP02], operators discussions ([LS97] or [IIS98], e.g.), further algorithmic development [CO07] and many other research activities.

3.4.3 Symbolic Regression

In short, symbolic regression is the induction of mathematical expressions on data. The key feature of this technique is, as Keijzer summarized in [Kei02], that the

[7]In fact, there are 2,554,416 possible programs with length 11, but only 12 (i.e., 0.00047%) of them are successes. For programs of length 14 this ratio is approximately 0.0007%, for bigger program sizes (up to 200 – 500) it levels off between 0.0001% and 0.0002% [LP98].

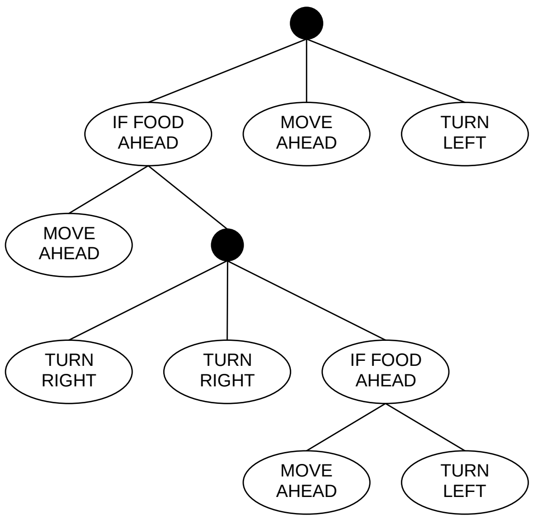


Figure 3.14: Santa Fe trail solution.

object of search is a symbolic description of a model, not just a set of coefficients in a pre-specified model. This is in sharp contrast with other methods of regression, including linear regression, polynomial approaches, or also artificial neural networks (ANNs), where a specific model is assumed and often only the complexity of this model can be varied.

The main goal of regression in general is to determine the relationship of a dependent (target) variable t to a set of specified independent (input) variables x. Thus, what we want to get is a function f that uses x and a set of coefficients w such that

$$t = f(x, w) + \epsilon \tag{3.1}$$

where ϵ represents the error (noise) term.

The form of f is usually pre-defined in standard regression techniques as for example linear regression (f_{LinReg}) and ANNs (f_{ANN}):

$$f_{LinReg}(x, w) = w_0 + w_1 x_1 + \ldots + w_n x_n \tag{3.2}$$

$$f_{ANN}(x, w) = w_0 \cdot g(w_1 x) \tag{3.3}$$

In linear regression, w is the set of coefficients $w_0, w_1, \ldots, w_n$. In ANNs we usually use an auxiliary transfer function g (which normally is a sigmoid function as for example the logistic function $\frac{1}{1+e^{-t}}$); the coefficients w are here called weights and include the weights from the hidden nodes to the output layer (w_0) and those from the input nodes to the hidden nodes (w_1) [Kei02].

In contrast to this, the function f which is searched for is not of any pre-specified form when applying genetic programming to symbolic regression. Instead, low-level functions are used and combined to more complex formulas during the GP process. Given a set of functions $f_1, \ldots, f_u$, the overall functional form induced by genetic programming can take a variety of forms. Usually, standard arithmetical functions such as addition, subtraction, multiplication, and division are in the set of functions $\mathbf{f}$, but also trigonometric, logical, and more complex functions could be included.

An exemplary composed function therefore could be:

$$f(x,w) = f_1(f_4(x_1), f_5(x3, w1), f_4(f_2(x_1, w_2)), x_2) \tag{3.4}$$

or, by filling in some concrete functions for the abstract symbols $\mathbf{f}$ and $\mathbf{w}$ we could get:

$$f_1(x) = +(*(0.5, x), 1) \equiv 0.5 * x + 1 \tag{3.5}$$

$$f_2(x) = +(2, *(x, x)) \equiv 2 + x * x \tag{3.6}$$

When it comes to evaluating solution candidates in a GP based symbolic regression algorithm, the formulae have to be evaluated on a certain set of evaluation data $\mathbf{X}$ yielding the estimated values $\mathbf{E}$. These estimated values are then compared to the original values $\mathbf{T}$, i.e. those which are known from data retrieval (experiments) or calculated applying the original formula to $\mathbf{X}$.
For example, let f_{target} be the target function

$$f_{target}(x) = -(*(0.5, *(x, x)), 2) \equiv 0.5 * x^2 - 2 \tag{3.7}$$

and the functions f_1 and f_2 solution candidates. Furthermore, let the input data $\mathbf{X}$ be

$$X = [-5, -4, \ldots, +4, +5]. \tag{3.8}$$

Thus, by evaluating f_{target}, f_1, and f_2 on $\mathbf{X}$ we get T, E_1, and E_2:

$$T = [10.5, 6, 2.5, 0, -1.5, -2, -1.5, 0, 2.5, 6, 10.5] \tag{3.9}$$

$$E_1 = [-1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5] \tag{3.10}$$

$$E_2 = [27, 18, 11, 6, 3, 2, 3, 6, 11, 18, 27] \tag{3.11}$$

By crossing f_1 and f_2, these become parent functions (*parent formula 1* and *2*) and we could for example get the *child formula* f_3:

$$f_3(x) = +(*(0.5, *(x, x)), 1) \equiv 0.5 * x * x + 1 \tag{3.12}$$

and by evaluating it on $\mathbf{X}$ we get E_3:

$$E_3 = [13.5, 9, 5.5, 3, 1.5, 1, 1.5, 3, 5.5, 9, 13, 5] \tag{3.13}$$

Graphical displays of the formulae f_1, f_2, and f_3 (labeled as parent and child functions) and their evaluations are given in the Figures 3.16 and 3.15, respectively.

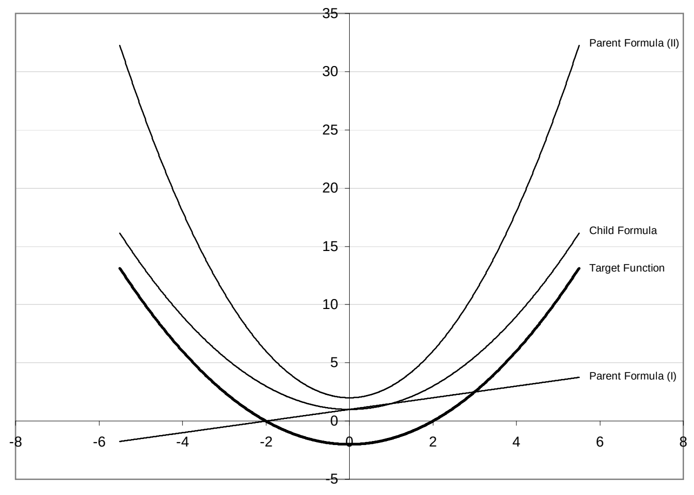


Figure 3.15: Symbolic regression example.

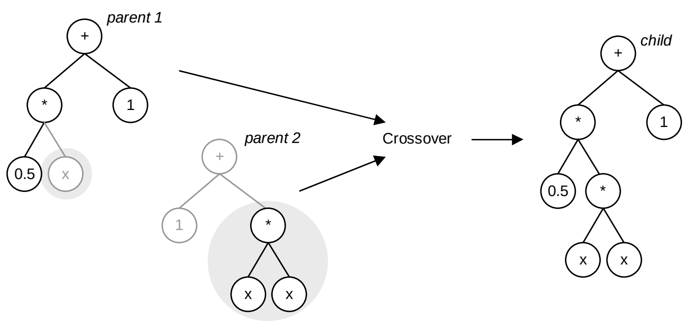


Figure 3.16: Exemplary formulae.

The task of GP in symbolic regression thus is to find a composition of the functions, input variables, and coefficients that minimizes the error of the function with respect to the desired target values. There are several ways how to measure this error, one of the simplest and probably most frequently used ones being the mean squared error (mse) function; the mean squared error of the vectors A and B each containing n values is calculated as

$$mse(A,B) = \frac{1}{n} * \sum_{k=1}^{n}(A_k - B_k)^2;\ |A| = |B| = n \tag{3.14}$$

So, we can calculate the fitness of f_1, f_2, and f_3 as $mse(E_1,T)$, $mse(E_2,T)$, and $mse(E_3,T)$, respectively yielding

$$\begin{aligned} fitness(f_1) &= 26.0 & (3.15)\\ fitness(f_2) &= 100.5 & (3.16)\\ fitness(f_3) &= 9.0 & (3.17) \end{aligned}$$

Whereas the search for formulas that minimize a given error function (or maximize some other given fitness function) is the major goal of GP based regression, the shape and the size of the solution could also be integrated into the fitness estimation function. The number and values of coefficients used is another issue that is tackled in the optimization process; the search process is also free whether to consider certain input variables or not, thus it is able to perform variables selection (possibly leading to dimensionality reduction) [Kei02].

3.4.4 Other GP Applications

Finally we shall here give a short list of problems for which GP has proven to be able to produce high quality results - this list of course comes without the claim of completeness.

Koza can be for sure seen as one of the pioneers of applying GP to a variety of different problems: In [Koz92], [Koz94], [KIAK99], and [KKS+03a] he reports (together with co-authors) on the GP based solving of problems for example in classification, regression, pattern recognition, computational molecular biology, emergent behavior, cellular automata, sorting networks, design of topology and component sizing for complex hardware structures (such as analog electrical circuits, controllers, and antenna), and many others. Many of those results can be considered human-competitive results, some even being patentable new inventions created by GP.

In hardware design, for example, one of the problem situations explained in [KIAK99] is the automated design of amplifiers. In general, an amplifier is a

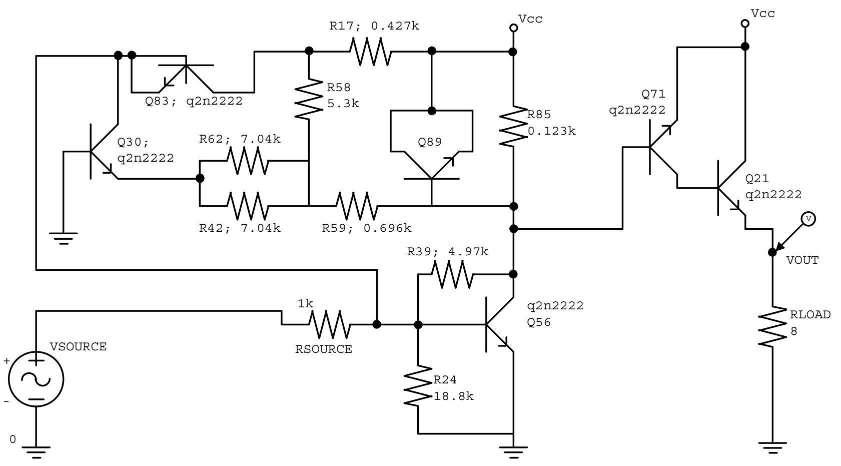


Figure 3.17: Design of a 10 dB Amplifier, created by GP ([KIAK99], Fig. 42.4).

circuit with one input and one output which multiplies the voltage of its input signal by a certain factor (the so-called voltage amplification factor) over a specified range of frequencies. The goal now is to realize such an amplifier only using resistors, capacitors, inductors, transistors and power sources; the functions set used thus includes component creating functions for creating digital gates, inductors, transistors, power supplies, and resistors. Solution candidates are tree structures representing complete hardware entities which can be displayed in a way which we are used to: Figure 3.17 resembles Figure 42.4 of [KIAK99] showing the best-of-run circuit realizing a 10 dB amplifier.

Of course, there is a vast number of other fields of applications for genetic programming. Numerous applications of GP to problems of practical and scientific importance have for example also been documented in the conference proceedings of the GECCO, CEC or EuroGP conferences ([CPFD$^+$03a], [CPFD$^+$03b], [DPB$^+$04a], [DPB$^+$04b], [BOA$^+$05], [KCA$^+$06], [TBB$^+$07], [KOL$^+$04], [KTC$^+$05], [CTE$^+$06], or [EOE$^+$07], e.g.). Please see the GP bibliography (Section 3.8) for a short list of sources of publications on those.

3.5 GP Schema Theories

As we have now summarized *how* genetic programming works, we shall now turn our minds towards investigations *why* it works so well. Holland's work in the mid-1970s produced the well-known GA schema theorem; schemata have since then been frequently used to demonstrate how and why GAs work. In fact, as is summarized in [PMR04], in the 1990s interest in GA theory shifted towards exact microscopic Markov chain models possibly with aggregated states. However, after the work of Stephens and collaborators in the late 1990s on exact schema theories based on the notion of dynamic building blocks and the connection highlighted by Vose between his model and a different type of exact schema-based model, it is now clear that Markov-chain and schema-based models are, when exact, just different representations of the same thing.

Genetic programming theory has had an, as Poli et. al. stated in [PMR04], "difficult childhood": After some early works on approximate GP schema theorems, it took quite some time until schema theories could be developed that give exact formulations for expected frequencies of schemata at the next generation.

In this section we give a rough overview of these GP schema theorems: After summarizing early work on GP schema theories in Section 3.5.1, which see schemata as components of programs, we give an introduction to rooted tree GP schema theories (Section 3.5.2) and an exact GP schema theory (Section 3.5.3). Finally, in Section 3.5.4 we summarize the GP schema theory concept.

The classification of schemata given in this section follows the grand concepts of [LP02], Chapters 3–6.

3.5.1 Program Component GP Schemata

First attempts to explain why GP works were given by Koza; in short, he gave an informal argument showing that Holland's schema theorem would work also for GP as described in [Koz92], pp. 116–119. In Koza's definition, a schema is defined as a set of program subtrees (S-expressions); a schema can so be used for defining a subspace of the program trees search space by collecting all programs that include all subtrees given by the schema. For example, the schema `H`=[(+ x 3), y] includes the programs `*`(y,+(x,3)) and `*`(+(y,3),+(2,+(x,3))) as they both include (at least) one occurrence of the S-expressions `(+` x 3) and `y`. This example is displayed graphically in Figure 3.18.

The first probabilistic model of GP that can be considered a mathematical formulation of a schema theorem for GP [LP02] was given by Altenberg in [Alt94a]. Also assuming very large populations, the neglection of mutation, and the application of proportional selection, he was able to calculate the frequency of a program at the next generation. Altenberg used a schema concept in which schemata are

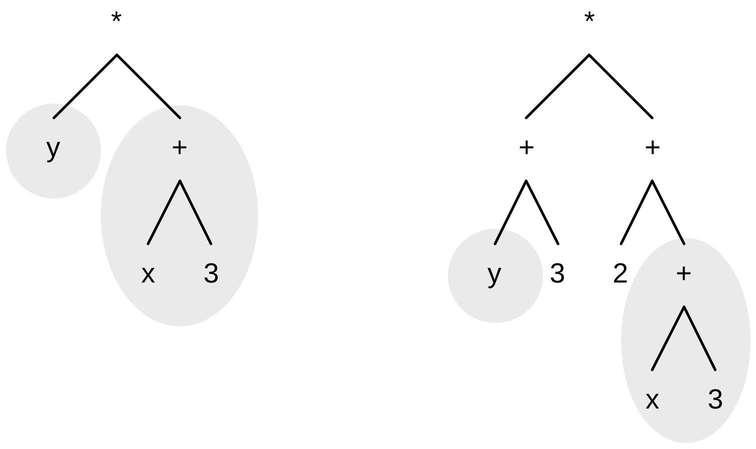


Figure 3.18: Programs matching the exemplary schema H=[(+ x 3), y].

subexpressions and not, as in Koza's work, collections of subexpressions.

O'Reilly formalized Koza's work on schemata ([O'R95], [OO94]) and derived a schema theorem for GP with proportional selection and crossover (but without mutation). The main difference to Koza's approach was that she defined schemata as collections of subtrees and tree fragments; tree fragments in this context are trees with at least one leaf being a "don't care" symbol ('#'). O'Reilly was also able to calculate the frequency of a program at the next generation; unfortunately, the frequency depends on the shape, the size and the composition of the trees containing the schemata investigated. Thus, frequencies are given rather as lower bounds than as concrete values. As O'Reilly argued in the discussion of her result, no hypotheses can be made on the basis of this theorem regarding the real propagation and the use of building blocks in GP.

Another approach was investigated by Whigham: He produced a definition of schemata for context free grammars and the related schema theorem which was published for example in [Whi95], [Whi96b], and [Whi96a]. Based on his definition of schemata he was able to give equations for the probabilities of disruption of schemata by crossover and mutation. Like in O'Reilly's work, also in Whigham's theorem the propagation of the components of schemata from one generation to the next is described.

In all these early attempts GP schemata were used for modeling how components (or groups of components) propagate within the population and how the number of these instances can vary over time.

3.5.2 Rooted Tree GP Schema Theories

In rooted tree GP schema theory, a schema can be seen as a set of points of the search space that share some syntactic feature. This can be defined in the following way [PMR04]: Let F be the set of functions used, and T the set of terminals. Syntactically, a **GP schema** is then defined as a tree composed of functions from the set $F \cup \{=\}$ and terminals from $T \cup \{=\}$; the primitive $=$ here means "don't care" and stands for a single terminal or function. Semantically, H is the set of programs that have the same shape and the same labels for the non-"=" nodes as the tree representation of H.

A simple example is given in Figure 3.19: Be $F = \{+, -, *\}$ and $T = \{x, y, z\}$, and the schema H given as $*(=, = (x, =))$. For example, the programs $*(y, *(x, x))$, $*(z, +(x, z))$, and $*(x, -(x, z))$ are program members of H, i.e. they are included in H's semantics.

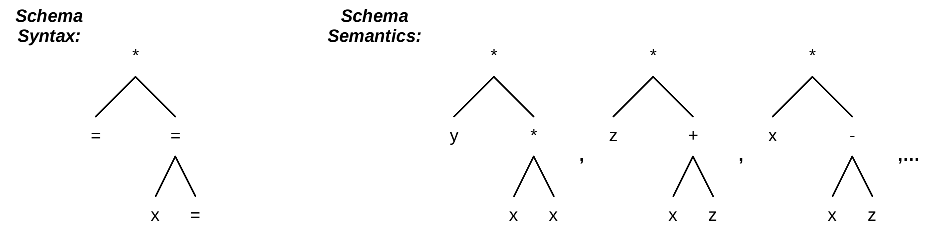


Figure 3.19: The rooted tree GP schema $*(=, = (x, =))$ and three exemplary programs of the schema's semantics.

Rosca proposed this kind of schemata in [Ros97] (using the symbol '#' instead of '='). He formulated his schema theorem so that it became possible to calculate a lower bound for a schema's frequency at the next generation. As a matter of fact, here also schemata divide the space of programs into subspaces containing programs of different sizes and shapes.

Contrary to this, the following fixed-size-and-shape theory for GP was developed by Poli and Langdon ([PL97c], [PL97a]):

Under the assumption that fitness proportional selection is applied, the probability of a program h sampling the schema H to be selected is

$$Pr\{h \in H\} = \frac{m(H,t)f(H,t)}{M\overline{f}(t)} \tag{3.18}$$

where $m(H, t)$ denotes the number of programs matching the schema H at generation

t, $f(H,t)$ the mean fitness of programs matching H, M the population size, and $\overline{f}(t)$ the mean fitness of the programs in the population.

The main idea is that the probability of the disruption of a schema can be estimated. Let $D_c(H)$ be the event "H is disrupted when a program h matching H is crossed over with a program $\hat{h}$"; as is described in full detail in cite [LP02], the probability of such a disruption caused by one-point crossover can be formulated as

$$Pr\{D_c(H)\} \leq p_{diff}(t)\left(1 - \frac{m(G(H),t)f(G(H),t)}{M\overline{f}(t)}\right)$$

$$+\frac{\mathcal{L}(H)}{N(H)-1}\frac{m(G(H),t)f(G(H),t) - m(H,t)f(H,t)}{M\overline{f}(t)} \tag{3.19}$$

where $G(H)$ is the shape of all programs matching the schema H (which is called the *hyperspace* of H), and $\mathcal{L}(H)$ the defining length of H; p_{diff} is the probability of the disruption of schema H by crossing h (matching H) with program $\hat{h}$ that has different shape than h, i.e. which is not in $G(H)$: $p_{diff}(t) = Pr(D_c(H)|\hat{h} \notin G(H))$.

When it comes to point mutation, a schema H will survive mutation only if all of its $\mathcal{O}(H)$ defining nodes are not modified. Thus, the probability of H being disrupted by mutation $Pr\{D_m(H)\}$ is dependent on the probability of a node to be altered (p_m):

$$Pr\{D_m(H)\} = 1 - (1 - p_m)^{\mathcal{O}(H)} \tag{3.20}$$

The overall formula uses these partial results and finally gives the expected number of programs matching schema H at generation $t+1$:

$$E[m(H,t+1)] \geq MPr\{h \in H\}(1 - Pr\{D_m(H)\})(1 - p_{xo}Pr\{D_c(H)\}) \tag{3.21}$$

By substituting (3.18), (3.19) and (3.20) in (3.21) we get the final overall formula for the lower bound of individuals sampling H at generation $t+1$ in generational GP with fitness proportional selection, one-point crossover and point mutation as it is given in [LP02].

This GP schema theorem, produced by generalizing Holland's GA schema theorem, thus gives a pessimistic lower bound for the expected number of copies of a schema in the next generation. In the next chapter we will summarize an exact GP schema theory, produced by generalizing an exact GA schema theorem and using the concept of hyperschemata.

3.5.3 Exact GP Schema Theory

In the previous section we have summarized pessimistic GP schema theory based on generalization of Holland's GA schema theorem. As Langdon and Poli summarize in [LP02], the usefulness of these schema theorems has been widely criticized (see [CP94], [Alt94b], [FG97], [FG98] or [Vos99], e.g.). In order to overcome its main drawbacks, namely that they are pessimistic and only give lower bounds for the expected numbers of instances for a given schema at the next generation, more exact schema theorems for GAs and GP had to be developed. These are going to be summarized in this section: After explaining the main idea of Stephen and Waelbroeck's GA schema theory, the hyperschema concept is summarized, and finally, on the basis of these hyperschemata, exact GP schema theorems.

An exact GA schema theorem has been developed by the end of the last millennium ([SW97], [SW99]): The total transmission probability α of a schema H is defined so that $\alpha(H,t)$ is the probability that at generation t the individuals of the GA's population will match H. Assuming a crossover probability p_{xo}, $\alpha(H,t)$ is calculated as:

$$\alpha(H,t) = (1-p_{xo})p(H,t) + \frac{p_{xo}}{N-1}\sum_{i=1}^{N-1} p(L(H,i),t)p(R(H,i),t) \tag{3.22}$$

with $L(H,i)$ and $R(H,i)$ being the left and right parts of schema H, respectively, and $p(H,t)$ the probability of selecting an individual matching H to become a parent. The "left" part of a schema H is thereby produced by replacing all elements of H at the positions from the given index i to N with "don't care" symbols (with N being the length of the bit strings); the "right" part of a schema H is produced by replacing all elements of H from position 1 to i with "don't care". The summation sums over all positions from 1 to $N-1$, i.e. over all possible crossover points. A generalization of this theorem to variable-length GAs has also been constructed [SPWR02].

After the publication of this exact GA schema theory, immediately the question came to mind whether it would be possible to extend pessimistic GP schema theories towards an exact GP schema theorem [LP02]. In fact, it was: Poli developed an exact GP schema theorem (see [Pol99a], [Pol00c], [Pol00b], [Pol00a], e.g.), a theorem which was then generalized by Poli and McPhee to become known as Poli and McPhee's Excact GP Schema Theorem ([PM01b], [PM01a], [Pol01], [PM03a], [PM03b], [PMR04], and [LP02]).

Assuming equal size and shape for GP programs, (3.22) can be also used for describing the transmission probability of a fixed-size-and-shape GP schema. In the presence of one-point crossover, the transmission probability for a GP schema H at generation t, $\alpha(H,t)$, can be thus given as

$$\alpha(H,t) = (1-p_{xo})p(H,t) + \frac{p_{xo}}{N(H)}\sum_{i=1}^{N-1} p(l(H,i),t)p(u(H,i),t) \tag{3.23}$$

with $l(H,i)$ and $u(H,i)$ being the lower and upper parts (building blocks) of schema H, respectively, and $N(H)$ the number of nodes in the schema (which is assumed to have the same size and shape as all other programs in the population). $l(H,i)$ is defined as the schema produced by replacing all nodes above cutting point i with "don't care" symbols, and $u(H,i)$ as the schema produced by replacing all nodes below cutting point i with "don't care" symbols. In analogy to (3.22), the summation in (3.23) sums over all possible crossover points.
Exemplary l and u schemata for the schema H = +(*(=,x),=) are shown in Figure 3.20.

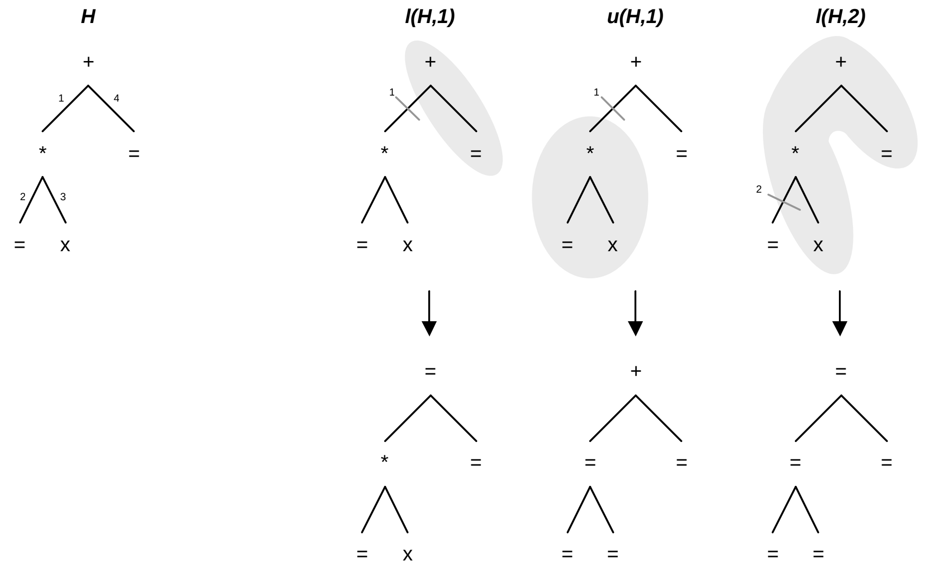


Figure 3.20: The GP schema H = +(*(=,x),=) and exemplary u and l schemata. Cross bars indicate crossover points, shaded regions show that parts of H which are replaced by "don't care" symbols.

In order to generalize this exact GP schema theorem so that it can be applied to populations of programs of different sizes and shapes, a more general schema approach is used, namely the GP hyperschema concept.

A **GP hyperschema** represents a set of schemata in the same way as a schema represents a set of program trees (which is why it is called "hyperschema"). This can be defined in the following way [PMR04]: Let F be the set of functions used, and T the set of terminals. Syntactically, a GP schema is then defined as a tree composed of functions from the set $F \cup \{=\}$ and terminals from $T \cup \{=, \#\}$. The primitives = and # here mean "don't care"; = stands for exactly one node, whereas # stands for any valid subtree.

Examples are shown in Figure 3.21: Be $F = \{+, -, *\}$ and $T = \{x, y, z\}$, and the hyperschema H given as $*(\#, = (x, =))$. The three exemplary programs $*(y, *(x, *))$, $*(*(x, y), +(x, z))$ and $*(*(*(x, y), y), +(x, z))$ are a part of H's semantics.

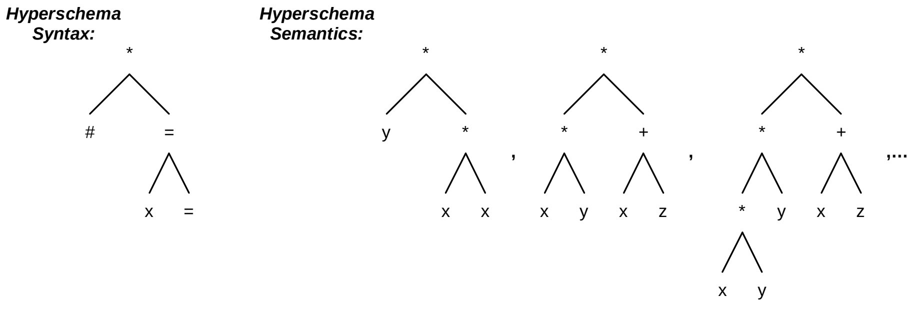


Figure 3.21: The GP hyperschema $*(\#, = (x, =))$ and three exemplary programs that are a part of the schema's semantics.

In analogy to $l(H, i)$ and $u(H, i)$ defined above and sketched in Figure 3.20, the hyperschemata building blocks $L(H, i)$ and $U(H, i)$ are defined in the following way: $L(H, i)$ is the hyperschema obtained by replacing all nodes on the path between crossover point i and the root of hyperschema H with = nodes, and all subtrees connected with those nodes with # nodes. $U(H, i)$ is the hyperschema obtained by replacing the subtree below crossover point i with a # node. [PMR04]

As examples might here also help to make this concept clearer, Figure 3.22 shows an exemplary schema H = +(*(=,x),=) and potential hyperschema building blocks. As for example shown in the second column, $L(H, 1)$ is constructed by turning all nodes between crossover point 1 and the root (in this case only the root node) into = nodes, and all subtrees of the so modified nodes become # nodes. $U(H, 1)$ is in column 3 constructed by replacing the subtree under crossover point 1 into a # node. And finally, as can be seen in column 4, $L(H, 2)$ is again constructed by turning all nodes from crossover point 2 to the root into = nodes, and all subtrees of the so modified nodes become # nodes.

Using hyperschemata, it is possible to formulate a general, exact GP schema theorem for populations of programs of any size or shape. The total transmission probability of a fixed-size-and-shape GP schema H is, for GP with one-point

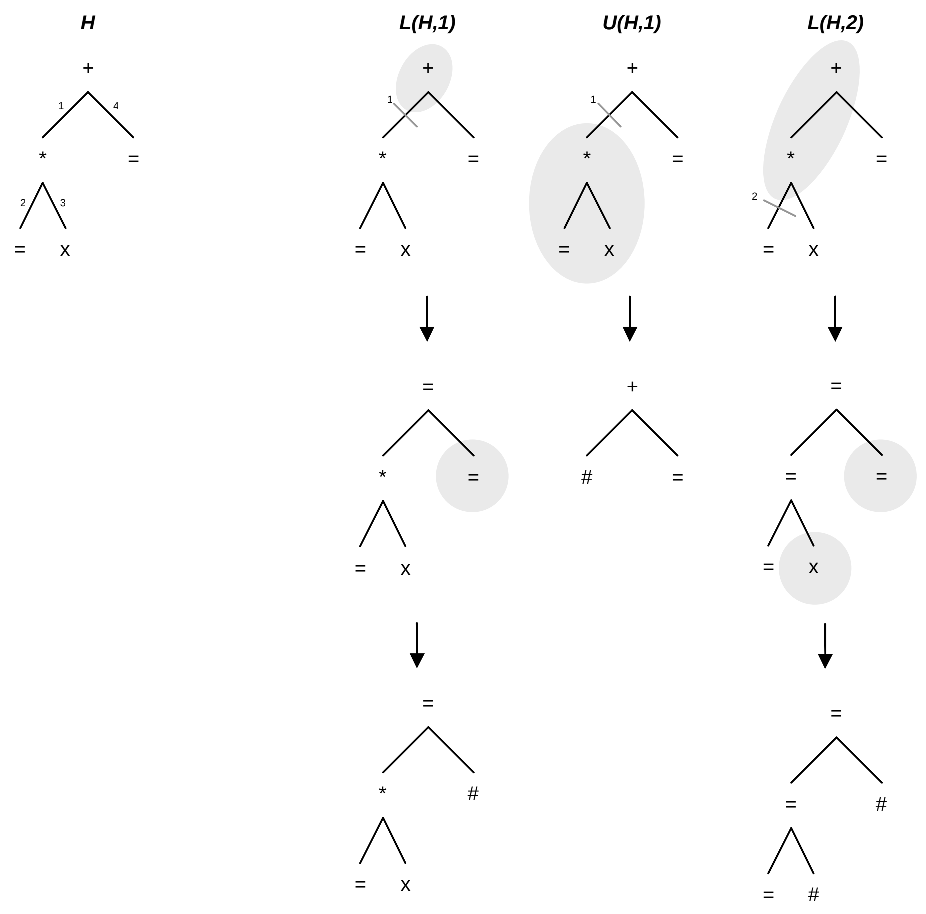


Figure 3.22: The GP schema $H = +(*(=, x), =)$ and exemplary U and L hyper-schema building blocks. Cross bars indicate crossover points, shaded regions show that parts of H which are modified.

crossover and no mutation, given as

$$\alpha(H,t) = (1 - p_{xo})p(H,t) + \tag{3.24}$$

$$p_{xo} \sum_{h_1} \sum_{h_2} \frac{p(h_1,t)p(h_2,t)}{\mathrm{NC}(h_1,h_2)} \sum_{i \in C(h_1,h_2)} \delta(h_1 \in L(H,i))\delta(h_2 \in U(H,i))$$

where $\mathrm{NC}(h_1, h_2)$ is the number of nodes in the tree fragment representing the common region of the programs h_1 and h_2, $C(h_1, h_2)$ is the set of indices of the crossover points in the common region of h_1 and h_2, and $\delta(x)$ is a function that returns 1 if x is true and 0 otherwise. The first two summations sum over all individuals in the population, i.e. we sum over all possible pairs of programs; the second summation sums over all indices of crossover points of the common region of

the respective programs pair.
This GP schema theorem is called the **Microscopic Exact GP Schema Theorem** in the sense that it is necessary to consider each member of the population.

Via several transformations and lemmata (which are not given here) it is finally possible to formulate the **Macroscopic Exact GP Schema Theorem**:

$$\alpha(H,t) = (1 - p_{xo})p(H,t) + \tag{3.25}$$

$$p_{xo} \sum_j \sum_k \frac{1}{\mathrm{NC}(G_j, G_k)} \sum_{i \in C(G_j, G_k)} p(L(H,i) \cap G_j, t) p(U(H,i) \cap G_k, t))$$

where $G(H)$ denotes the schema that is obtained by replacing all nodes in a schema H by "don't care" symbols[8]; the sets $L(H,i) \cap G_j$ and $U(H,i) \cap G_k$ are either schemata (of fixed size and shape), or the empty set $\emptyset$.
Thus, using this theorem (3.5.3), it is at last possible to give the exact transmission probability of a schema for genetic programming under one-point crossover and no mutation; an exact schema theorem for GP is established. We have here omitted lots of transformation steps and proofs; for these, the interested reader is for example referred to [PM03a], [PM03b], [LP02], or [PMR04].

An overview of the development of approximate and exact schema theorems for GAs and GP is graphically shown in Figure 3.23 (as given in [PMR04]).

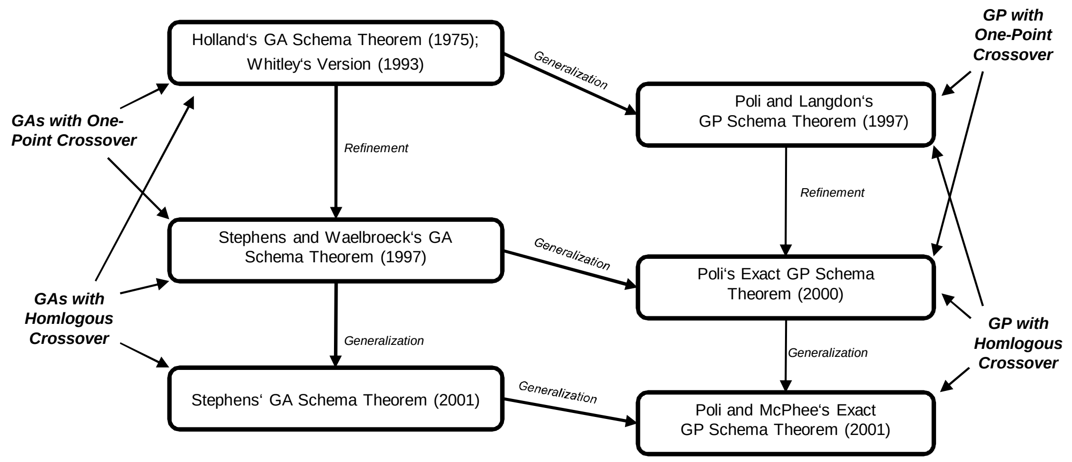

Holland's GA Schema Theorem (1975); Whitley's Version (1993)
Generalization
Poli and Langdon's GP Schema Theorem (1997)
GP with One-Point Crossover
GAs with One-Point Crossover
Refinement
Refinement
Stephens and Waelbroeck's GA Schema Theorem (1997)
Generalization
Poli's Exact GP Schema Theorem (2000)
GAs with Homlogous Crossover
Generalization
Generalization
GP with Homlogous Crossover
Stephens' GA Schema Theorem (2001)
Generalization
Poli and McPhee's Exact GP Schema Theorem (2001)

Figure 3.23: Relation between approximate and exact schema theorems for different representations and different forms of crossover (in the absence of mutation).

[8]$G(H)$ is called the *hyperspace* of H.

3.5.4 Summary

Until the development of the GP schema theorems described in this section, GP theory was typically considered scarce, approximate and not terribly useful [PM01c]. Especially the facts, that GP is relatively young and that building theories for variable size structures are very complex, are considered the reasons for this.

Significant breakthroughs, which have been summarized in this section, have fundamentally changed this understanding; after the development of GP schema theorems, we now have an exact GP theory based on schema and hyperschema concepts.

3.6 Current GP Challenges and Research Areas

Of course, theoretical work on GP was by far not finished after the development of GP schema theorems. Even though they shall be not be discussed in detail here, we still want to line out a selection of current research areas in GP theory.

For example, operators design for GP has been discussed in numerous publications; extensive analysis of initialization, crossover and mutation operators can be found in [Lan99], [ES03] or [LN00], for example.

The genetic programming search space has been subject to theoretical analysis (see [LP98], [LP02], e.g.). Experimental exploration of the GP search space by random sampling can be used for comparing GP to random search or other search techniques. Additionally, hypotheses have been stated regarding minimum and maximum tree depth.

As has already been mentioned before, a Markov model for GAs has been formulated by Vose, see [NV92], [VL91] and [Vos99] for explanations. In short, a GA is modeled as a Markov chain; selection, mutation, and crossover are incorporated into an explicitly given transition matrix, thus the method is complete, and no special assumptions are made which restrict populations or population trajectories.
This GA Markov model could also be extended to GP using the schema GP theory described in the previous section, which gives exact formulas for computing the probability that reproduction and recombination create any specific program. A GP Markov chain model is then easily obtained by plugging this ingredient into a minor extension of Vose's model of GAs [PMR04]; in fact, an alternative approach for describing the dynamics of evolutionary algorithms is provided by this theory.

One fact has been known for genetic programming since some of its first applications and has been frequently reported: Programs in genetic programming populations tend to grow in size ([Ang94], [Lan95], [NB95], [SFD96], [AA05], [Ang98], [TH02]). "Redundancy", "introns" and, probably most frequently used as well as with the most negative connotation, "bloat" have (amongst others) been used since then as names for this tendency. In principle, it means that introns, i.e. code which has no effect on the performance of the program containing it, grow during the GP process; it is in fact a phenomenon also known from natural evolution [WL96].

Of course, this seems to be an unwanted phenomenon and is not conform to "Occam's Razor", a law attributed to the 14th-century Francisian friar William of Ockham. This law is also known as the "law of parsimony", the Latin principle "entia non sunt multiplicanda praeter necessitatem" meaning that "entities should not be multiplied beyond necessity" is also often quoted. In principle, this law demands the selection of exactly that theory that postulates the fewest entities and introduces the fewest assumptions (of course, in case if there are multiple competing theories which are considered equal in other respects). Argumentations pointing out how and why GP does or does not fulfill Occam's law can be found in [Dro98] and [LP97], e.g.[9]

Examples for bloat are given in Figure 3.24: In the left example, the left subtree will always return $(x - (0 * y + x)) = x - x = 0$ and since the multiplication of 0 with any other value always results in 0, the result of the whole program will always be 0 regardless of the values of x, y and z. In fact, the whole right subtree becomes code that does not influence the whole program's evaluation. In the second example shown on the right part of Figure 3.24, A will always be smaller than $A+4$, thus the condition of the root condition will always be fulfilled and "else"-branch will never be activated.

In contrary to the examples in Figure 3.24, in which bloat is rather obvious, there are also of course examples in which it can be seen that GP will not always automatically produce rather simple results. For example, in Figure 3.17 we have shown the design of a 10dB amplifier created by GP [KIAK99]. After the excision of everything except voltage gain and selected other parts[10], a strongly simplified circuit is designed (shown in Figure 3.25); when retested, as is documented in [KIAK99], this excised circuit proves to be an amplifier with a gain almost identical to that

[9]Especially "The Myth of Occam's Razor" [Tho18], a paper written by Thorburn in 1918, is worth reading in this context as it discusses the origins of the principle. For more discussions on Occam's razor and its reception in philosophy and science the interested reader is referred to [Jac94], [Nol97], [Pop92] or [RF99].

[10]As documented in [KIAK99], all parts of the circuit were excised except the voltage gain and the so-called quasi-Darlington emitter-follower section.

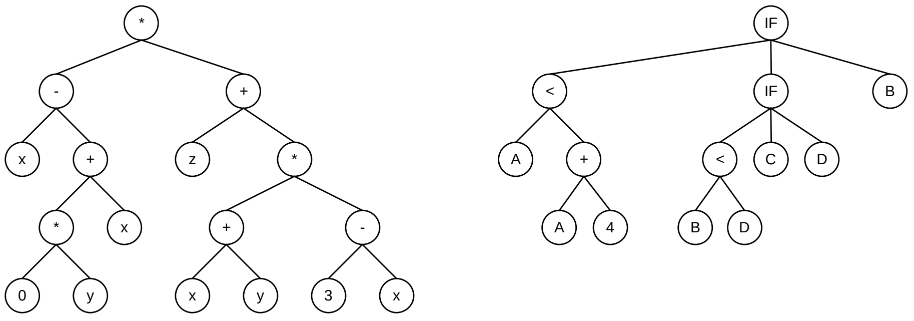


Figure 3.24: Examples for bloat.

of the original entire circuit. Even though this is not exactly what we see as bloat in that sense that there is (in the original circuit) code that is irrelevant, still it is obvious that there is a remarkable amount of code which is of almost no effect to the program's performance.

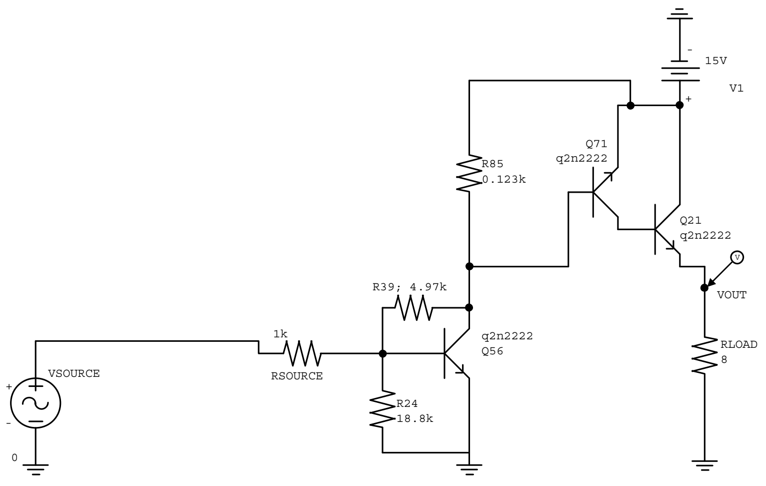


Figure 3.25: Design of a 10 dB Amplifier, created by GP, excised ([KIAK99], Fig. 42.8)

In their article entitled "Fitness Causes Bloat" [LP97], Langdon and Poli showed that fitness based selection seems to be responsible for the solutions' growth in size; fitness based parents selection therefore leads to code bloat. In this context bloat has also been ironically described as "survival of the fattest".

According to [Zha97], [Zha00] and [LP02], approaches used for preventing or at least decreasing bloat include, but are not restricted to the following anti-bloat techniques:

- Size and/or depth limitations: The growth of programs is limited, programs are not allowed to become bigger in size and/or depth (where the size of a program is normally the size of its structure tree and its depth the depth of its structure tree). Size limits are nowadays commonly used, see for example [KIAK99].
- Incorporation of program size in the selection process: An also often used technique to combat bloat is to include some preference for smaller programs in the criterion used to select programs for reproduction; this additional factor to selection is also called parsimony pressure. Examples and analysis can be for example found in [Kin93], [Zha97], [Zha00], [SF98] and [SH98].
- Incorporation of program size in evaluation: The size of a program could of course also be incorporated in its evaluation. It might also be included as one the goals which the GP population tries to reach ([LN00], [EN01]).
- Genetic operators: Besides selection and evaluation, several crossover and mutation operators have been proposed which are designed so that they combat bloating, see for example [Ang98], [PL97b] or [Lan00].

Often we see another tendency of GP that does not fulfill Occam's law, namely that it is prone to producing programs that are overspecified. This means that programs that are too complex for the problem at hand and that much simpler programs could fulfill the given task as well; especially in data based modeling this phenomenon is also known as "overfitting". We shall come back to this topic in Chapter 7.

Another field of GP research is the development of practical guides for ideal parameter settings for GP. As we find in [SOG04], for example, GP researchers and practitioners are often frustrated by the lack of theory available to guide them in selecting key algorithm parameters; GP population sizes, for example, run from ten to a million members or more, but at present there is no practical guide to knowing when to choose which size. [SOG04] here gives a population-sizing relationship depending on tree size, solution complexity, problem difficulty and building block expression probability.

Furthermore, numerous other theoretical topics are widely discussed in the GP community, lots of them directly connected to well known problems (or rather challenges) with GP. Selected ones are to be mentioned in the next chapters.

As a part of the conclusions of [LP02], Langdon and Poli demand that GP users might like to consider how their GP populations are evolving, whether they are converging and, if so, whether they are converging in the right direction. At the present, many GP packages offer only few possibilities to monitor populations. As we are going to demonstrate in later chapters, this is exactly what we try to accomplish by investigating dynamics in the populations of our GA and GP implementations.

3.7 Conclusion

In this chapter, genetic programming has been summarized and described as a powerful extension to the genetic algorithm. In fact, GP is more than a GA extension: It can be rather seen as the art of evolving computer programs and as a generic concept for the automated programming of computers.

After describing GP basics and a variety of applications for GP, we have summarized theoretical concepts for GP based on schemata and hyperschemata. Problems and challenges in the context of GP have also been discussed.

In the following chapters we shall now come back to algorithmic developments in GAs. These advanced concepts can of course also be used with GP. In Chapter 7 we then come back to GP and its application to data based system identification; we also demonstrate the effects of these algorithmic enhancements in GP.

3.8 Bibliographic Remarks

There are numerous books, journals, and articles available that survey the field of genetic programming. In the following we summarize some of the most important ones.

The following books are widely considered very important sources of information about GP:

- J. R. Koza et al.: *Genetic Programming I - IV* ([Koz92], [Koz94], [KIAK99], [KKS$^+$03a]): A series of books on theory and praxis of genetic programming by John Koza and varying co-authors
- W. Banzhaf et al.: *Genetic Programming – An Introduction* [BNKF98]
- W. Langdon: *Genetic Programming and Data Structures* [Lan98]
- W. Langdon and R. Poli: *Foundations of Genetic Programming* [LP02]

The following journals are dedicated to either theory and applications of genetic programming or evolutionary computation in general:

- *Genetic Programming and Evolvable Machines* (Springer Netherlands)
- *IEEE Transactions on Evolutionary Computation* (IEEE)
- *Evolutionary Computation* (MIT Press)

Moreover, several conference and workshop proceedings include papers related to genetic programming. Some examples are the following ones:

- *Genetic and Evolutionary Computation Conference (GECCO)*, a recombination of the *International Conference on Genetic Algorithms* and the *Genetic Programming Conference*
- *Congress on Evolutionary Computation (CEC)*
- *Parallel Problem Solving from Nature (PPSN)*
- *European Conference on Genetic Programming (EuroGP)*

Of course there is lots of GP-related information available on the internet including theoretical background and practical applications, course slides and source code. The probably most comprehensive overview of publications in GP is *The Genetic Programming Bibliography* which is maintained by Langdon, Gustavson, and Koza and available at http://www.cs.bham.ac.uk/ wbl/biblio/.

Finally, publications of the *Heuristic and Evolutionary Algorithms Laboratory (HEAL)* (including several articles on GAs and GP) are available at http://www.heuristiclab.com/publications/.

Chapter 4

Enhanced Selection Concepts

4.1 Gender Specific Parents Selection

Inspired by the idea of male vigor and female choice as it is considered in the model of sexual selection discussed in the area of population genetics, a new selection paradigm for GAs called SexualGA has been developed [WA05b]. The main idea of this gender specific selection scheme is to use two different selection schemes for the selection of the two parents required for each crossover. So it becomes possible to simulate the concept of male vigor and female choice by using random selection as the first selection scheme and another selection strategy with far more selection pressure as the second one (e.g. roulette wheel selection or linear rank selection).

This gender specific selection concept not only brings the concept of GAs a little bit more towards its biological archetype, but it also has relevant advantages compared to classical GA approaches particularly concerning flexibility. By using two different selection concepts simultaneously a GA user can influence the selection pressure level of a GA run more precisely. It is thus also possible to control the interplay between genetic diversity supporting and reducing forces in a more directed way and to better tune GA behavior depending on the individual needs of the attacked optimization problem. Further discussions and a comparison of solution qualities achieved using this principle can for example be found in [WA05b].

4.2 Offspring Selection

Offspring selection (OS, [AWW05a], [AWW05b]) considers not only the fitness of the parents in order to produce a child for the ongoing evolutionary process. Additionally, the fitness value of the evenly produced offspring is compared to the fitness values of its own parents. An offspring is accepted as a candidate for the further evolutionary process if and only if the reproduction and possibly the mutation operator were able to produce an offspring that could outperform the fitness of its own parents. This strategy guarantees that evolution is presumed mainly with crossover results that were able to mix the properties of their parents in an advantageous way.

As in the case of conventional GAs or GP, offspring are generated by parent selection, crossover, and mutation. In a second (offspring) selection step, the number of offspring to be generated is defined to depend on a predefined ratio parameter giving the quotient of next generation members that have to outperform their own parents (success ratio, *SuccRatio*). As long as this ratio is not fulfilled, further children are created and only the successful offspring will definitely become members of the next generation (as illustrated in Figure 4.1). When the postulated ratio is reached, the rest of the next generation's members is randomly chosen from the children that did not reach the success criterion.

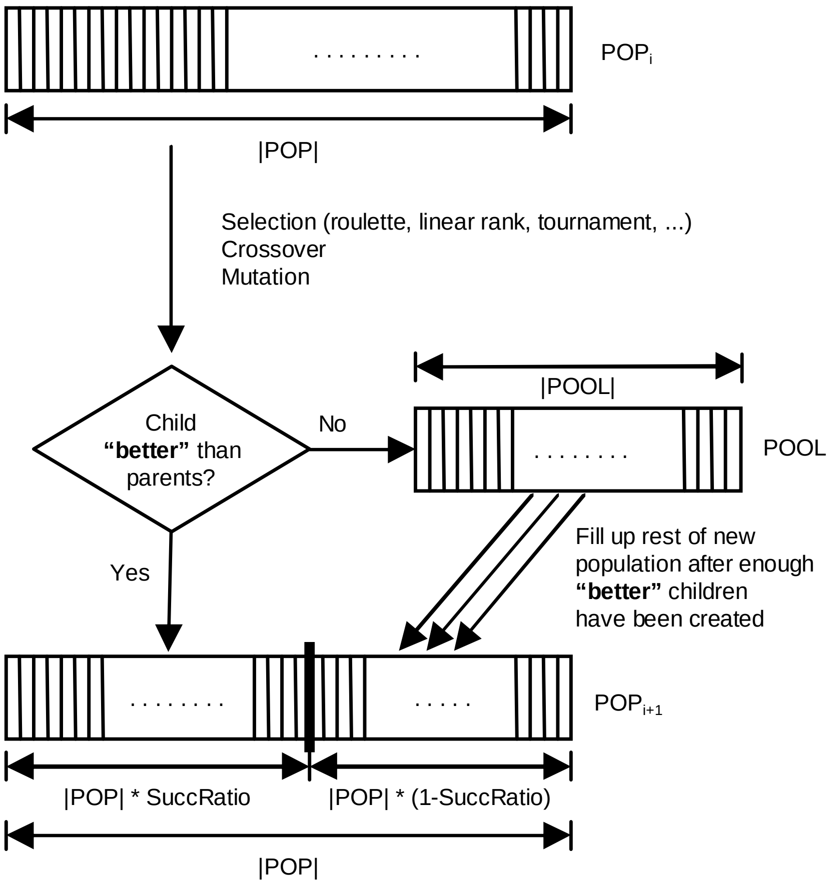
Within our new selection model, selection pressure *selPres* is a measure for the effort that is necessary to produce a sufficient number of successful solutions; it is defined as the ratio of generated candidates to the population size:

$$selPres = \frac{|virtualPOP| + |POP| \cdot SuccRatio}{|POP|} \tag{4.1}$$

where *virtualPOP* denotes the virtual population, the pool of solutions that are not considered immediately but might be inserted into the new population as "lucky losers". Figure 4.1 schematically displays the main aspects of this offspring selection model.

An upper limit for selection pressure gives a quite intuitive termination heuristics: If it is no more possible to find a sufficient number of offspring that outperform their parents, the algorithm terminates.

This offspring selection concepts plays a major role in the SASEGASA algorithm ([AW04]), a segregative genetic algorithm that incorporates aspects of simulated annealing and self adaptive selection pressure steering, see also Section 5.1.6.



. . . .
POP$_i$
|POP|
Selection (roulette, linear rank, tournament, ...)
Crossover
Mutation
|POOL|
Child **“better”** than parents?
No
. . . .
POOL
Fill up rest of new population after enough **“better”** children have been created
Yes
. . . .
. . . . .
POP$_{i+1}$
|POP| * SuccRatio
|POP| * (1-SuccRatio)
|POP|

Figure 4.1: Offspring selection.

Chapter 5

Parallel Concepts for Genetic Algorithms and Genetic Programming

5.1 Parallelization of Genetic Algorithms

When it comes to parallel and distributed approaches in metaheuristics or, in general, computer science, the basic idea is to divide a given task into subtasks and to solve those simultaneously using multiple processors. In GAs, this divide-and-conquer approach can be used in different ways; some methods for parallelizing GAs change the behavior of the GA whereas others do not. Some methods, especially fine-grained parallel GAs, are designed to exploit massively parallel computer architectures, while others (especially coarse grained parallel GAs) are better qualified for multi-computers with fewer and more powerful processing elements. Detailed descriptions and classification of distributed GAs can be found in [CPG99], [CP01], [CP97], [AT99] and [Alb05].

Roughly speaking (and following classifications as for example the one given in [DLJD00]), parallel GA concepts can be classified into global parallelization, coarse-grained parallel GAs and fine-grained parallel GAs. In practical applications, the most popular model is the coarse-grained model, also known as the island model.

5.1.1 Global Parallelization

Similar to the basic behavior of standard GAs, parallel GAs (PGAs) implementing the global parallelization concept have only one single panmictic population, i.e., all individuals have the chance to mate with any other individual of the population. In principle, the algorithm behaves as a standard GA and the global GA has exactly the same qualitative properties as a serial GA. In most applications the evaluation of the individuals is parallelized because the fitness of an individual is independent from the rest of the population and there is no need for communication during this phase.

The probably most frequently used concept in PGAs is the master-slave approach: One master node executes the GA (selection, crossover, and mutation), and the evaluation of the individuals is distributed among several slave processors. Communication is necessary only insofar as each processor receives its subset of individuals for evaluation and returns the fitness values after evaluation.

5.1.2 Coarse-Grained Parallel GAs

In coarse-grained PGAs, the population is divided into multiple sub-populations (also called islands or demes) that evolve more or less independently from each other and only occasionally exchange individuals; this exchange of individuals is called migration. In contrast to the global parallelization model, coarse-grained parallel GAs introduce fundamental changes in the structure of the GA and have a different behavior than serial GAs. Coarse-grained parallel GAs are also known as distributed GAs because they are usually implemented on distributed-memory computers; they are also often referred to as island parallel GAs in analogy to the island model known in population genetics, a model which considers relatively isolated demes.

The left part of Figure 5.1 schematically shows the design of a coarse-grained parallel GA: Each circle represents a simple GA, and there is (infrequent) communication between the populations. The qualitative performance of a coarse-grained parallel GA is influenced by the number and size of the certain demes and also by the information exchange between them (migration). The main idea of this type of parallel GAs is that relatively isolated demes will converge to different regions of the solution-space, and that migration and recombination will combine the relevant solution parts [SWM91]. However, at present there is only one model in the theory of coarse-grained parallel GAs, namely the SASEGASA algorithm (see Section 4.2), that considers the concept of selection pressure for recombining the favorable attributes of solutions evolved in the different demes. The concept of coarse-grained parallel GAs is the most frequently used parallel GA concept as those are quite easy to be implemented and a natural extension to the general concept of serial GAs making use of commonly available cluster computing facilities.

Research in the field of parallel island GA is still going on; for example, in [HLdVL07] the question whether island model is fault tolerant is discussed in detail.

5.1.3 Fine-Grained Parallel GAs

The fine-grained model for PGAs, sketched in the right part of Figure 5.1, considers a large number of very small demes; it defines one spatially distributed population, and is especially well-suited for massively parallel computers or any other supercomputing architecture.

5.1.4 Hybrid Parallel GAs

A recent research topic in the area of parallel evolutionary computation is the combination of certain aspects of the different population models resulting in so-called hybrid parallel GAs, of which most are coarse-grained at the upper level and fine-grained at the lower levels. Alternatively, it is also possible to use coarse-grained GAs at the high as well as at the low levels in order to force stronger mixing at the low levels using high migration rates and a low migration rate at the high level [CP97].

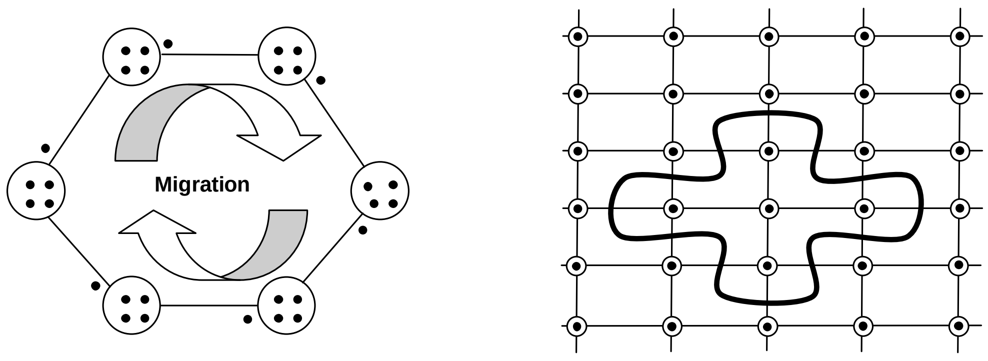


Figure 5.1: Schematic sketches of basic concepts for parallel genetic algorithms. Left: The coarse-grained parallel GA, right: The fine-grained parallel GA.

5.1.5 Migration

Especially for coarse-grained parallel GAs the concept of migration is considered to be the main success criterion in terms of global solution quality. The most important parameters for migration are:

- The communication topology which defines the interconnections between the subpopulations (demes),
- the migration scheme which controls which individuals (best, random) migrate from one deme to another and which individuals should be replaced (worst, random, doubles),
- the migration rate which determines how many individuals migrate, and
- the migration interval or migration gap that determines the frequency of migrations.

As was for example already summarized and explained in [Aff03] and [Aff05], the most essential question concerning migration is, when and to which extent migration should take place. Usually, migration occurs synchronously meaning that it occurs at predetermined constant intervals. Still, this approach is known to be slow and inefficient for some problems [AT99].

Asynchronous migration schemes perform communication between demes only after specific events. The migration rate which determines how many individuals undergo migration at every exchange can be expressed as a percentage of the population size or as an absolute value. The majority of articles in this field suggest migration rates between 5% and 20% of the population size. However, the choice of this parameter is considered to be very problem dependent [AT99].

5.1.6 The SASEGASA

Recent theory of self adaptive selection pressure steering (as explained in Section 4.2) plays a major role in defying the conventions of recent parallel GA theory. Within these models it becomes possible to detect local premature convergence, i.e. premature convergence in a certain deme, exactly if the actually required selection pressure exceeds an upper limit. Thus, local premature convergence can be detected independently in all demes, which should give a high potential in terms of efficiency especially for parallel implementations. Furthermore, the fact that selection pressure is adjusted self-adaptively with respect to the potential of genetic information stored in the certain demes, makes the concept of a parallel GA much more independent in terms of migration parameters.

Offspring selection plays a major role in the SASEGASA algorithm [AW04], a segregative genetic algorithm that incorporates aspects of simulated annealing and self adaptive selection pressure steering. This SASEGASA concept is also designed to retard the unwanted effects of premature convergence by combining concepts of GAs, ES and simulated annealing. Roughly speaking, premature convergence occurs when a GA's population reaches a suboptimal state in which the genetic operators can no longer produce offspring that are fitter than their parents (e.g. [Fog94])[1]. A self-adaptive feature of this approach is realized in that way that the number of individuals that have to be created in order to produce a sufficient amount of "successful" offspring depends on the actual stadium of the evolutionary process.

5.2 Parallel Genetic Programming

Of course, as GP is built on the basic ideas of genetic algorithms, any parallelization concept for GAs can also be applied for genetic programming. Koza, for example, mentions that the asynchronous island approach is the most commonly used approach to parallelization of genetic programming [KIAK99]; experiments documented in [AK96] and [KIAK99] indicate that a modest amount of migration (circa 2% of a processor's population in each of four possible directions) is better than extremely high or extremely low amounts of migration.
Parallel GP on a network of transputers has for example been described already in 1995 [AK95], and in [BKSS99] a parallel computer system that performs a half petaflop per day is described as well as island model based parallel GP using it. Massively parallel GP and respective application scenarios have been discussed in [JP96], e.g.

In order not to arouse confusion about multi-population genetic algorithms (or genetic programming) on the one side and multi-agent GP systems on the other side, we shall here mention some main aspects of these two approaches.
Multi population GAs have already been discussed: Several populations evolve independently, and depending on their organization (and level of interaction) they can be classified into global parallelization, coarse-grained parallel GAs and fine-grained parallel GAs. The coarse-grained model is considered the most popular model for

[1]Several methods have been proposed to combat premature convergence in genetic algorithms; some of them are described in-depth in [CG93], [Gol89] and [Jon75].

practical, it is also known as the island model.

When talking about multi-agent GP systems, normally a different approach is referred to: Evolving programs of one (or more) GP population(s) are seen as agent(s), each of which has a defined purpose [Lan98]. These agents show a rather high level of interaction with each other; programs can for example use other programs by calling them as sub-routines. Obviously, the ADF concept can be easily integrated into (or rather combined with) multi agent GP; ADFs can then be either shared by the agents or specific to each purpose / agent [LP02]. Application examples are also given in [KIAK99] and [KKS$^+$03a].

Chapter 6

Data Based Modeling and System Identification

6.1 Basics

Data mining is in general understood as the practice of automatically searching for patterns in large stores of data. Nowadays, incredibly large (and quickly growing) amounts of data are collected in commercial, administrative, and scientific databases; several sciences produce extreme amounts of information which are often collected automatically. This is why it is impossible to analyze and exploit all these data manually; what is needed are intelligent computer systems that can extract useful information (such as general rules or interesting patterns) from large amounts of observations. In short, “data mining is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data” [FPSS96].

One of the ways how genetic algorithms and, more precisely, genetic programming can be used in data mining is its application in systems analysis and data based system modeling: A given system is to be analyzed and its behavior modeled by a mathematical model, the process is therefore (especially in the context of modeling dynamic physical systems) called system identification [Lju99].

The principles have already been summarized in the GP introduction chapter, Section 3.4.3 on symbolic regression, and they shall be repeated and extended in the following:

The main goal of regression is to determine the relationship of a dependent (target) variable t to a set of specified independent (input) variables x. Thus, what we want to get is a function f that uses x and a set of coefficients w such that

$$t = f(x, w) + \epsilon \tag{6.1}$$

where ϵ represents the error (noise) term.

Applying this procedure we assume that a model can be created with which it will also be able to predict correct outputs for other data examples (test samples); from the training data we want to generalize to situations not known (or allowed to analyze) during the training phase.

When it comes to evaluating a model (a solution candidate in a GP based modeling algorithm, e.g.), the formula has to be evaluated on a certain set of evaluation (training) data **X** yielding the estimated values **E**. These estimated target values are compared to the original values **T** , i.e. those which are known from data retrieval (experiments) or calculated applying the original formula to **X**.
This comparison is done by calculating the error between original and calculated target values. There are several ways how to measure this error, one of the simplest and probably most frequently used ones being the mean squared error (mse) function; the mean squared error of the vectors A and B each containing n values is calculated as

$$mse(A, B) = \frac{1}{n} * \sum_{k=1}^{n} (A_k - B_k)^2 \tag{6.2}$$

Some of the major problems of in data based modeling are *noise* and *overfitting*:

- In common language, on the one hand we know noise as in general that what is heard, but on the other hand also as unwanted sound which is added to the audio signals that are of interest. Furthermore, the concept of noise is also known in image and video processing, where it is used more to describe unwanted signals that are rather disturbing. In the context of data based modeling we often see that additional and somehow unwanted values are added to the original signals; this disturbing additional data is called noise.

- In machine learning, overfitting is understood as the exceeding fitting of models to given data. As already mentioned, data based training of models is done using training data, i.e. sets of training examples of the functions which are searched for; the problem is that it can happen – especially in cases where too complex models are trained or the training process is executed too long – that the learner may adjust to very specific features or samples of the training data.

Even a structurally inadequate model may fit to given training data perfectly if the model is complex enough.
From the point of view of mathematical systems theory, we assume that a system Σ can be described by a function $\phi(\theta) : u \rightarrow y$, where u and y are the system's input and output, respectively, ϕ describes the structure of the function and θ denotes the vector of parameters. Data based structure identification is supposed to find a function $\psi(\hat{\theta}) : u \rightarrow y$ that reproduces the system's output. The more parameters are stored in $\hat{\theta}$ the easier it becomes to reproduce the given training data, but it also becomes more probable that $\psi(\hat{\theta})$ represents not the basic behavior of Σ but rather the measured signal (which also includes noise). Of course, as we do in general not know the size of θ (or the structure of ϕ), we cannot know when $\hat{\theta}$ becomes "too big".
Overfitting can also be seen as a violation of Occam's razor (see Section 3.6 for explanations on this); fitting too exactly to (noisy) training data might lead to a model whose ability to generalize is far worse than the general applicability of a simpler model.
Unfortunately there is no rule how to generally avoid overfitting as we often do not exactly know the complexity of the system whose behavior is to be modeled. However, there are several techniques that can help to avoid it: For instance, overfitting might cause a significant rise of the variances of the estimated parameter values $\hat{\theta}_i$, i.e. the parameter values estimated in independent identification runs diverge (which should of course not be the case if the structure of ψ and the size of $\hat{\theta}$ are correct); early stopping and the use of validation sets which are not included in the training data can also help to decrease the probability of overfitting.

Thus, accuracy (on training data) is not the only requirement for the result of the modeling process: Compact and (if possible) minimal models are preferred as they can be used in other applications easier. It is, of course, not easy to find that models that ignore unimportant details and capture the behavior of the system that is analyzed; due to this challenging character of the task of system identification, modeling has been considered as "an art" [Mor91].

In the following section we are going to explain the problems of noise and overfitting using a simple example.

6.2 An Example

Let us consider the following example: Be S a system whose behavior is to be modeled using the input / output (target) training examples given in Table 6.1 (where X and Y values denote input and output data, respectively).

Table 6.1: Data based modeling example: Training data.

X	Y	X	Y
-15	-1571.1605	-4	229.6581
-14	-1405.3919	-3	249.9523
-13	-644.6956	-2	518.4009
-12	-398.4149	-1	294.8873
-11	-69.9755	0	22.0334
-10	-87.4658	1	-193.7337
-9	126.4967	2	-146.7154
-8	227.3979	3	-294.5191
-7	309.4894	4	-179.5208
-6	522.4300	5	-353.2186
-5	474.8867		

By looking at the values as shown in Figure 6.1 the suspicion is aroused that there might be a cubic connection between the X and Y values, distorted by additive noise. This is in fact correct: The data were generated using the model $y = x^3 - 100x + 100$ and adding noise (uniformly distributed in the interval [-250; +250]). This is why the original function $x^3 - 100x + 100$ is also depicted in Figure 6.1.

If we want to evaluate the original formula that was used for simulating the system ($x^3-100x+100$), we can for example evaluate this model on all integral values for X in the range of the given training data (i.e., -15, -14, ..., 4, 5) and calculate the mean squared differences of these calculated values and the given training target data for Y which yields 18,556.4719 – the “fitness” of the original formula therefore is approximately 18,556.

Now let us suppose that we do not know or suspect anything about the system or its order. We could therefore try for example polynomial approaches of order 2, 3, 10 and 20; thus, we assume model structures of the form

$$y = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n \tag{6.3}$$

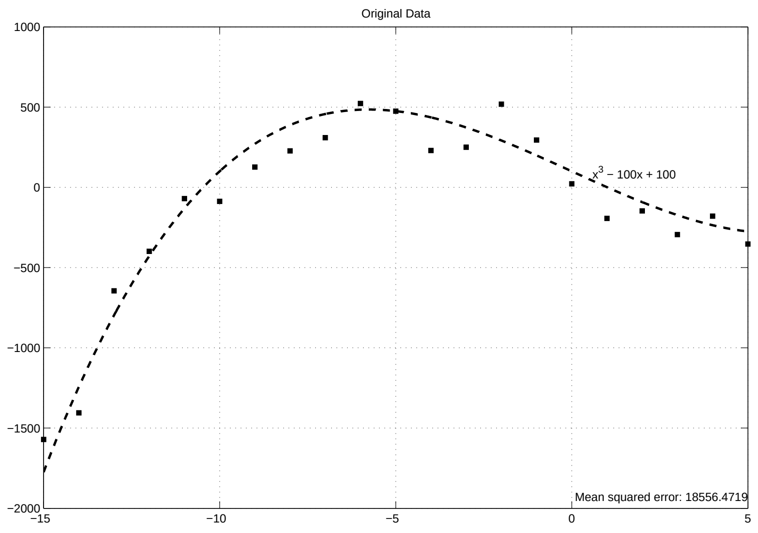


Figure 6.1: Data based modeling example: Training data.

for a model of order n. The parameters $[a_0, a_1, a_2, \ldots, a_n]$ are now to be set so that the model fits the given training data as exactly as possible[1].

As we see in the Figures 6.2, 6.3, 6.4 and 6.5, the quadratic model performs fairly, the model of order 3 performs better on the given training data, and the models of order 10 and especially 20 perform even a lot better; the polynomial of order 20 is even able to explain the training data perfectly. The quality of the so generated models of order 1, 3, 10 and 20 is approximately 244,218, 14,435, 6,605 and 0[2], respectively.

Now let us assume that test data is available for evaluating the models; this test data is not included in the training data but rather used for estimating the quality of the models produced (and of the identification method itself). These test data are given in Table 6.2.

Now we see that the linear model performs even worse on the test data ($mse_{test} \approx 25 * 10^6$, see Figure 6.6); the cubic model, which performed a lot better in training, is much more accurate also on test data ($mse_{test} \approx 4.5 * 10^6$, see Figure 6.7).

[1]The source code for this calculation of optimal models of order 2, 3, 10 and 20 is given in the appendix of this chapter, Section 6.5. There the reader can also find the source code used for generating the exemplary training and test data sets used here.

[2]Minor inaccuracies are here due to numerical imprecisions.

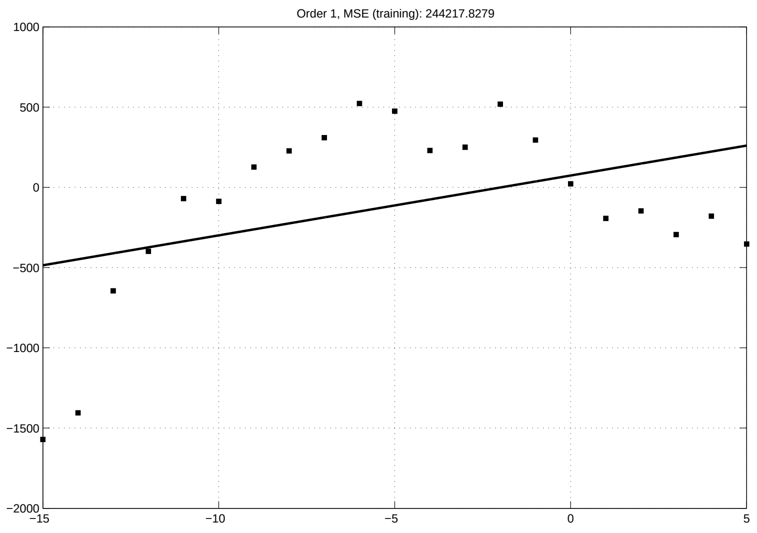


Figure 6.2: Data based modeling example: Evaluation of an optimally fit linear model.

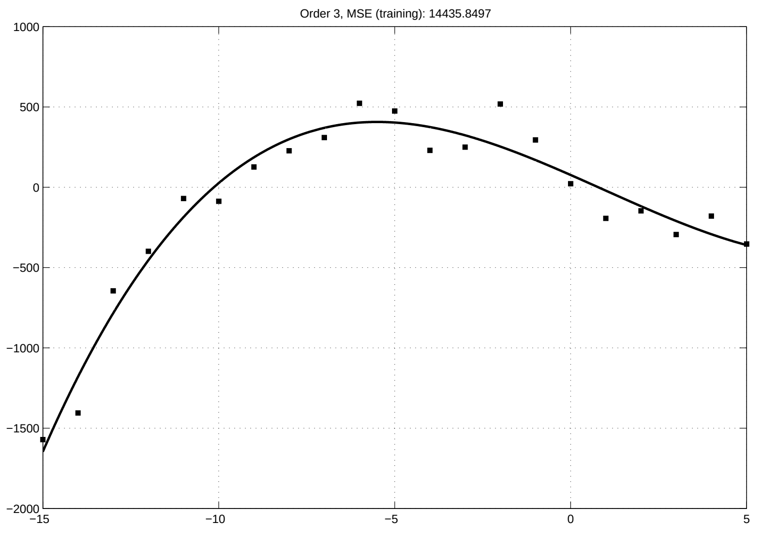


Figure 6.3: Data based modeling example: Evaluation of an optimally fit cubic model.

So, does this trend go on and does thus better fit on training data guarantee better fit on test data? Analyzing the test performance of the models of order 10

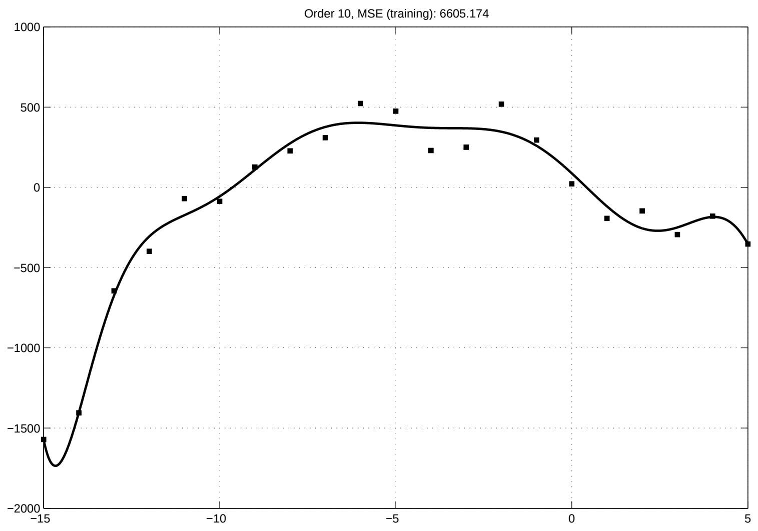


Figure 6.4: Data based modeling example: Evaluation of an optimally fit polynomial model ($n = 10$).

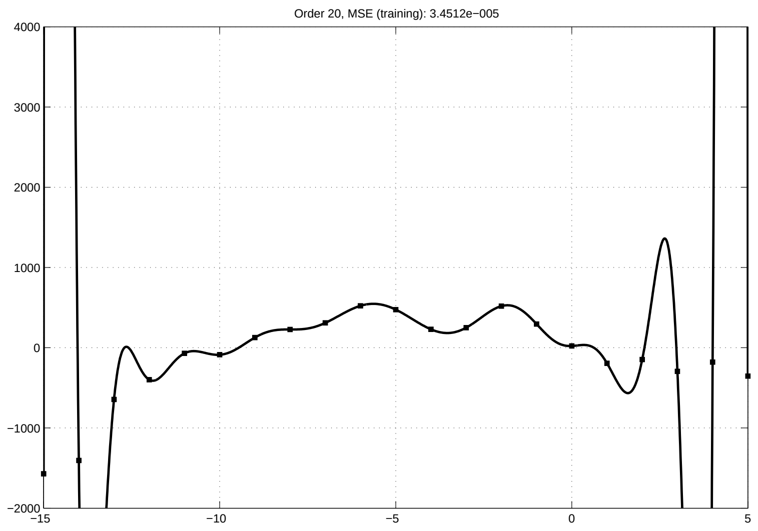


Figure 6.5: Data based modeling example: Evaluation of an optimally fit polynomial model ($n = 20$).

Table 6.2: Data based modeling example: Test data.

X	Y	X	Y
6	-381.4362	16	2609.0386
7	-73.1285	17	3147.7311
8	-226.3715	18	3941.3802
9	60.7464	19	5006.4839
10	-84.9143	20	5957.1595
11	251.8633	21	7424.0707
12	877.4408	22	8664.473
13	1149.4064	23	9937.4536
14	1666.7466	24	11452.5263
15	1941.3963	25	12980.5208

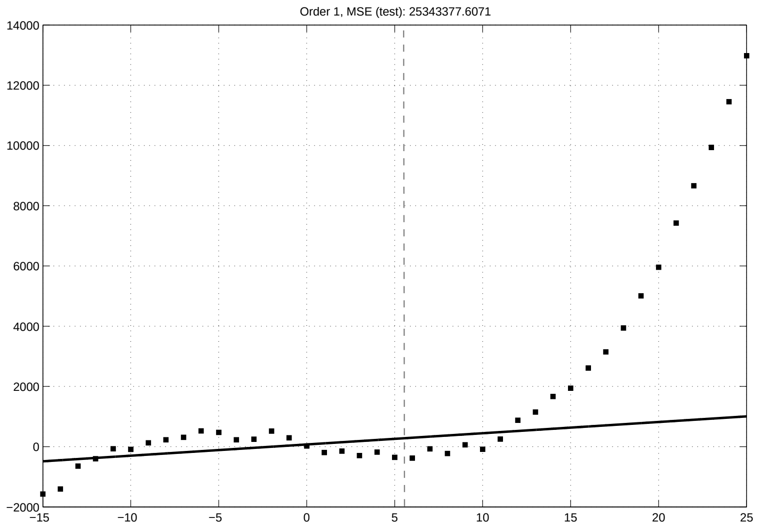


Figure 6.6: Data based modeling example: Evaluation of an optimally fit linear model on training and test data.

and 20 the answer to this question obviously is: No. In Figure 6.8 we see that the polynomial model of order 10 predicts something completely out of the range of the given test data yielding mean squared error value of $5 * 10^{16}$; the evaluation of the model of order 20 is not shown, its mean squared error on test data is $5.8 * 10^{34}$.

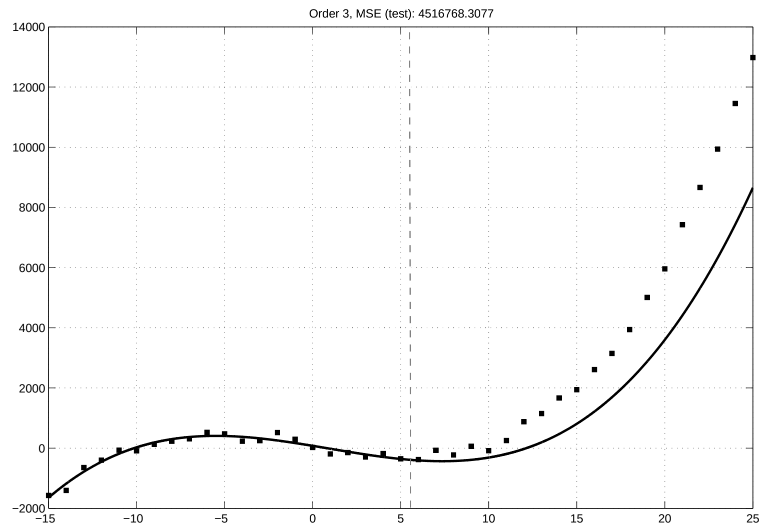


Figure 6.7: Data based modeling example: Evaluation of an optimally fit cubic model on training and test data.

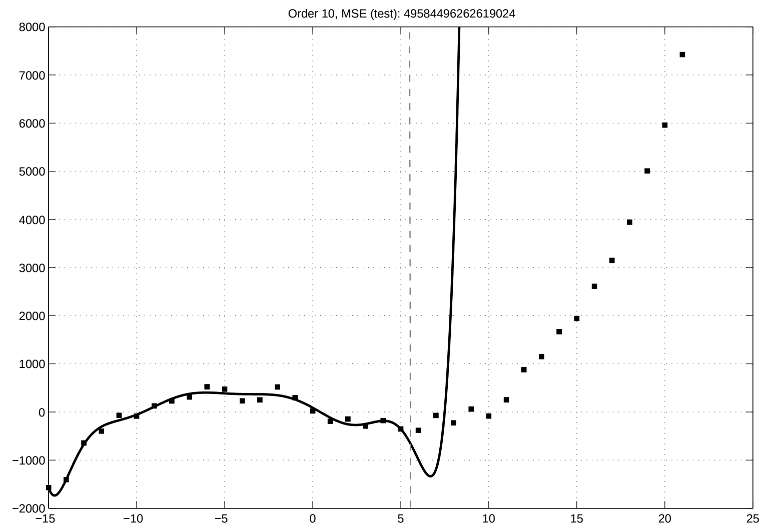


Figure 6.8: Data based modeling example: Evaluation of an optimally fit polynomial model ($n = 10$) on training and test data.

Summarizing this example we give an overview of training and test errors for the data and models mentioned above in Figure 6.9 (models of order 0 and 5 were created in the same way as the other models). This behavior is often observed: As the number of parameters increases, often the training errors tend to decrease; in the beginning, test errors are also likely to decrease[3], but after some time (as soon as overfitting happens), test errors start to increase with increasing training effort. Please note that the training and test errors shown in Figure 6.9 are depicted on a logarithmic y-axis.

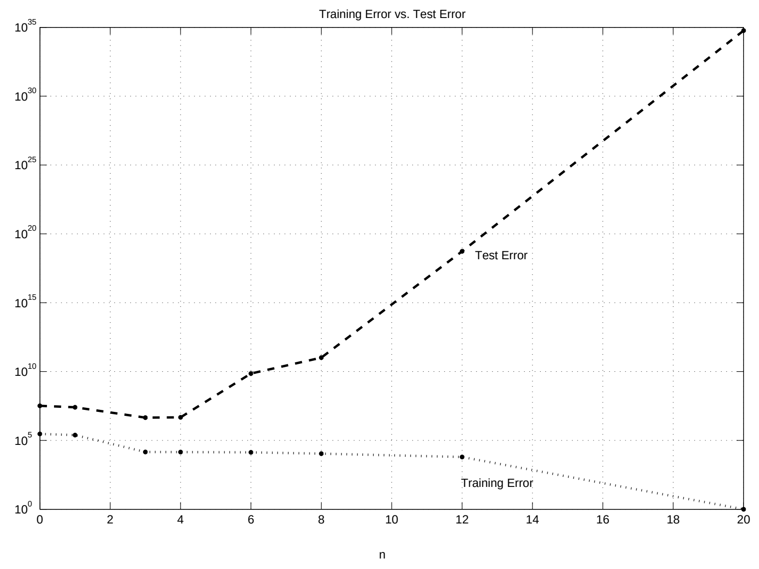


Figure 6.9: Data based modeling example: Summary of training and test errors for varying numbers of parameters n.

[3]In the summary chart displayed in Figure 6.9 we have intentionally omitted the training and test errors for $n = 2$. The reason is that it would have shown that in this particular case the test error for the quadratic model is a lot worse than for the linear as well as the cubic model; this would be correct, of course, but it so it is easier to sketch the characteristic behavior of first decreasing and then increasing test errors as the number of parameters increases.

6.3 The Basic Steps in System Identification

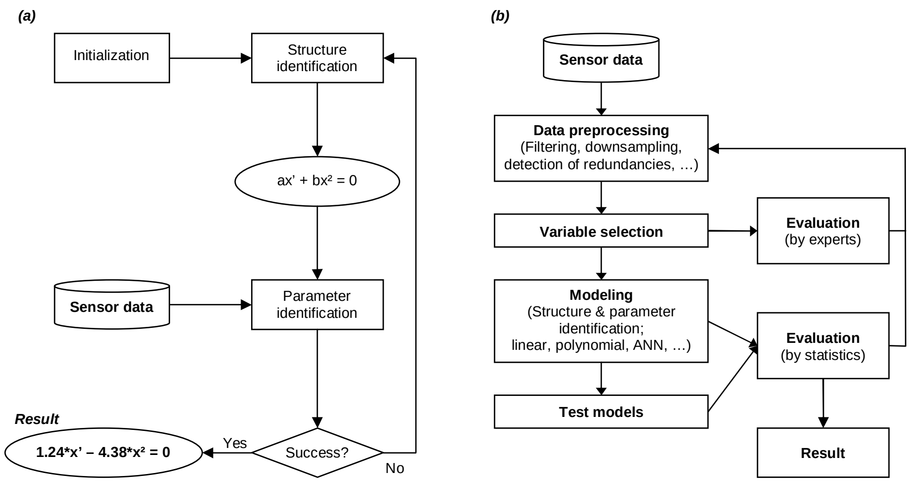


Figure 6.10: The basic steps in system identification ([BES01], [WEdR06]).

The following two phases in data based modeling are often distinguished: Structural identification and parameter optimization.

- First, structural identification is hereby seen as the determination of the structure of the model for the system which is to be analyzed; physical knowledge, for example, can influence the decision regarding the mathematical structure of the formula. This of course includes the determination of the functions used, the order of the formula (in the case of polynomial approaches, e.g.) and, in the case of dynamical models, potential time lags for the input variables used. In the simple example given previously this step was the decision to use a polynomial modeling approach; for example, the decision to try a polynomial model $y = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ of specific orders was the structural identification part. As we tried several polynomials of different orders we simply executed the procedure several times; this is exactly what is indicated by the feedback loop in Figure 6.10 (a).

- Parameter identification is then the second step: Based on training data, the parameters of the formula are determined (optimized) meaning that the coefficients and, if used, time lags are fixed. Basically, this is what we did in the previous example by calculating the coefficients for the polynomials of different orders separately.

This separation is schematically shown in the left part (a) of Figure 6.10 (adapted from [BES01]).

Of course, the whole process of building models out of data includes more steps than those mentioned above. Especially data preprocessing is a very important issue, i.e. preparing data before it is used for the "real" modeling process. For example, as we have proposed in [WEdR06], data downsampling, filtering and the removal of data without information should be applied in order to retrieve preprocessed data on which it is easier to efficiently generate appropriate models.
Variables selection is also often considered a key issue in data based modeling: Those variables are selected from the pool of variables which shall be used for the essential modeling process. For example, variables which do not include information (since they are constant in the whole data set, e.g.) or are redundant to other ones can be omitted for simplifying the modeling process. Variables selection can be thereby done using expert knowledge or statistical methods. Exhaustive statistical methods are available as well as sequential iterative forward or backward variable selection:

- Exhaustive search is executed by computing all possible combinations of variables and evaluating them; exactly that combination of variables will be selected which provides best approximation of measurement data. This method is able to provide an optimal solution (if the process is linear), but especially for higher dimensional problems (including big numbers of variables) it requires excessive computation time. In order to overcome this drawback, forward and backward selection can be used as alternatives even if they provide only sub-optimal solutions.

- In sequential forward selection the algorithm sequentially derives the list of input variables. In the first step, only one input variable is considered where that variable is selected that minimizes the sum of squared errors. In the next step, another input variable is selected where once again that variable is chosen which minimizes the sum of squared errors; the algorithm iteratively adds more and more input variables until a predefined accuracy is reached and hence the algorithm terminates. Of course the results depend on the chosen basis functions.

- The main difference when applying backward selection is that the algorithm starts with all variables available as set of selected variables and then iteratively removes variables that do not have a statistically measurable connection with the observed (measured) target values.

- Hybrid variants combining backward selection and a subsequent forward selection step have also been investigated for producing good results very efficiently.

These basic steps of the data driven modeling process are shown in the right part (b) of Figure 6.10.

As we see in both diagrams shown in Figure 6.10, the total system identification process based on measurement data is not finished as soon as models are created. A decision whether the model at hand is appropriate and fulfills the given quality requirements has to be made during a subsequent validation step. If this validation (often also called test phase[4]) fails, the process might be repeated starting again at the structural identification or data preprocessing step.

The major drawback of this classical approach is obvious: As the structure of the model has to be fixed before identifying parameters, thus it has to use *a priori* knowledge. However, there is a large number of applications in which the a priori model information is not available to the desired precision. For all these cases, several generic so-called "model free" approaches are widely used, ranging from simple static maps up to self-organizing neural networks. For a critical discussion of ANN based identification of a diesel engine's NO_x emissions see for example [dRLF$^+$05], [PP01] for a specific spectral analysis tool to describe the behavior of a plant or [THL94] for a neural network approach.

In spite of the evident simplicity of generic approaches, the drawbacks are known as well: Over-parameterization, lack of extrapolation and often even of interpolation capabilities [dRLF$^+$05], large data requirements etc. All these problems are related to the fact that these approaches essentially reorganize the data, but nothing more.

6.4 Data Based Modeling Using Genetic Programming

Using genetic programming for data based modeling brings along the advantage that we are able to design an identification process that automatically incorporates variables selection, structural identification and parameters optimization in one process.

The function f which is searched for is not of any pre-specified form when applying genetic programming to data based modeling; during the GP process, low-level functions are combined to more complex formulas. Given a set of functions $f_1, \ldots, f_u$, the overall functional form induced by genetic programming can take a

[4]Please note that in some cases the terms validation and test phase are used synonymously, but often (and also in the following test case documentations) the validation and test phase are are separate model analysis phases. Detailed explanation is to come in the following sections.

variety of forms. Usually, standard arithmetical functions such as addition, subtraction, multiplication, and division are in the set of functions **f**, but also trigonometric, logical, and more complex functions can be included.

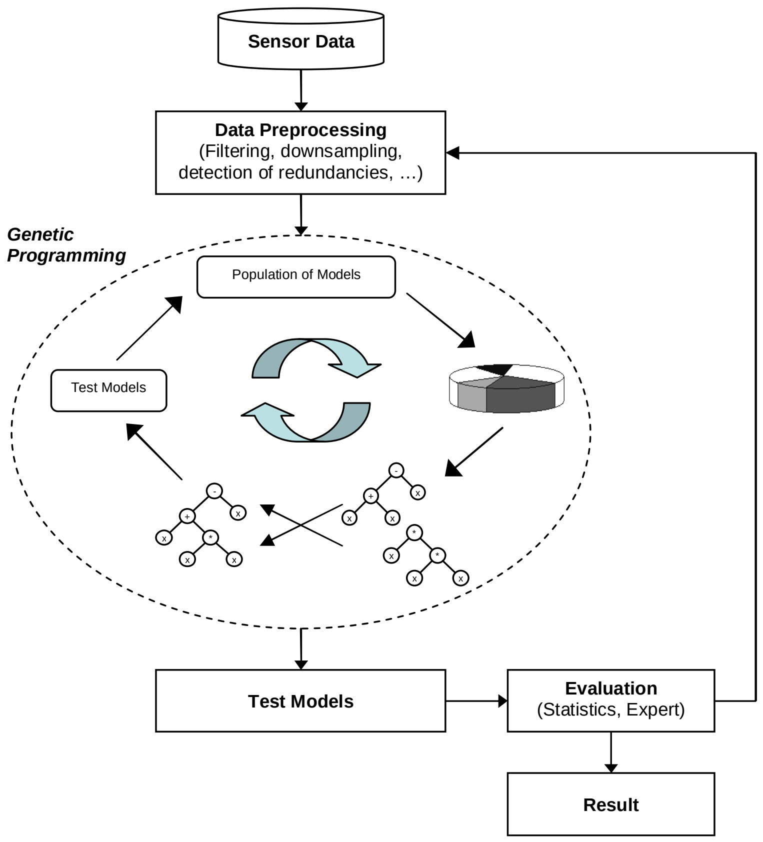


Figure 6.11: The basic steps of GP-based system identification.

Thus, the key feature of this technique is that the object of search is a symbolic description of a model, not just a set of coefficients in a pre-specified model. This is in sharp contrast with other methods of regression, including linear regression, polynomial approaches, or also artificial neural networks, where a specific model structure is assumed and often only the complexity of this model can be varied. Of course, data preprocessing and a separate validation / test phase are also parts of the GP-based modeling process; the main workflow is sketched in Figure 6.11.

In the following chapters we shall see, how data based modeling using genetic programming has been implemented in HeuristicLab (HL): After giving a general description of the basics of this GP implementation in Chapter 7, application domains and respective specific aspects implemented in HL are discussed in Chapter 8.

6.5 Appendix: Fitting Polynomials to Data

In Section 6.2 we have given a simple example for the basic steps in data based modeling. On the basis of given training and test data we have demonstrated basic concepts such as structural identification, parameter identification, and training and test evaluation of models. A polynomial approach is demonstrated, i.e. all models are of the form

$$y = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n \tag{6.4}$$

and the task is to determine optimal values for n and $a_0, a_1 \ldots a_n$.

For the fitting of a polynomial of order n we first compose a matrix M as a concatenation of the input values (namely the x values of the training data, i.e. all values in $[-15; 5]$) potentiated by $0, 1, \ldots, n$:

$$Z = [X^0 X^1 \ldots X^n], X^k = [x_1^k x_2^k \ldots x_N^k]^T \tag{6.5}$$

where X^k is a column vector consisting of all N input values to the power of n.

Secondly, the training target values Y are, after transposing the matrices, divided by Z using the right matrix division function (`/`); this numerically solves the system of linear equations defined by the order of the model n, the input data Z and the target values Y. Thus, we get the coefficients $a_0, a_1 \ldots a_n$ in the result of this division (as a vector p) and calculate the estimated target values $\hat{Y}$ (in the source code denoted as `Yhat`) by multiplying *poly* and Z; this represents the evaluation of the identified polynomial for each given sample.

The training and test qualities are calculated using the mean squared errors function, i.e. we calculate the sum of squared residuals and divide by the number of samples considered. The data documented in Section 6.2 were generated using a noise range of 500.

All data generation and modeling steps used in Section 6.2 were implemented in MATLAB©, Version 7.0 (R14). In the following we give the essential parts of the source code that was used for generating the data used and fitting the models to the given training data. The random generator is initialized in the beginning so that the results are comparable in each execution.

```
function StructureIdentificationExample(n, TrainingStart,
TrainingEnd, TestStart, TestEnd, NoiseRange)
%Creates training and test data for the specified ranges:
%- One input X, one output Y.
%- Y = X^3 - 100X + 100 + noise
%- Uniformly distributed noise is added
%A polynomial modeling approach is executed for order n; parameters are fit
%to training data using the right matrix division command.
%Results are displayed graphically including model quality information (MSE).

% -- preparations -----------

if nargin<6, NoiseRange = 500, end; if nargin<5, TestEnd = 25, end;
if nargin<4, TestStart = 6, end; if nargin<3, TrainingEnd = 5, end;
if nargin<2, TrainingStart = -15, end; if nargin<1, n = 5, end;

rand('state',12345);

X = TrainingStart:TrainingEnd; for i=1:length(X)
    Y(i,1)  = X(i)^3 - 100*X(i) + 100;
    Y_(i,1) = rand(1)*NoiseRange - NoiseRange/2;
    Y(i,1)  = Y(i,1) + Y_(i,1);
end Z = zeros(length(X), n+1); for j = 1 : (n+1)
    for i=1:length(X)
        Z(i,j) = X(i)^(j-1);
    end
end

Y = Y'; Z = Z';

x = TrainingStart:0.01:TrainingEnd; z = zeros(length(x), n+1); for j
= 1 : (n+1)
    for i=1:length(x)
        z(i,j) = x(i)^(j-1);
    end
end z = z';

% -- training -----------

poly = Y/Z; Yhat = poly*Z; error = Yhat-Y; mse =
sum(error.*error)/length(Y)

y = poly*z;

figure plot(X,Y,'s', 'MarkerEdgeColor','k', 'MarkerFaceColor','k',
'MarkerSize',5) hold on; plot(x,y,'k', 'LineWidth',2); hold off;
grid on; titlestr = ['Order ' num2str(n) ', MSE (training): '
num2str(mse)]; title(titlestr);

% -- test -----------

Xtest = TestStart:TestEnd; for i=1:length(Xtest)
    Ytest(i,1)  = Xtest(i)*Xtest(i)*Xtest(i) - 100*Xtest(i) + 100;
    Ytest_(i,1) = rand(1)*NoiseRange - NoiseRange/2;
    Ytest(i,1)  = Ytest(i,1) + Ytest_(i,1);
end

Ztest = zeros(length(Xtest), n+1); for j = 1 : (n+1)
    for i=1:length(Xtest)
        Ztest(i,j) = Xtest(i)^(j-1);
    end
end

Ytest = Ytest'; Ztest = Ztest'; YhatTest = poly*Ztest; errorTest =
YhatTest-Ytest; mseTest = sum(errorTest.*errorTest)/length(YhatTest)

x = TrainingStart:0.01:TestEnd; z = zeros(length(x), n+1); for j = 1
: (n+1)
    for i=1:length(x)
        z(i,j) = x(i)^(j-1);
    end
end z = z'; y = poly*z;

figure plot([X Xtest],[Y Ytest],'s', 'MarkerEdgeColor','k',
'MarkerFaceColor','k', 'MarkerSize',5) hold on; plot(x,y,'k',
'LineWidth',2); hold off; grid on; titlestr = ['Order ' num2str(n)
', MSE (test): ' num2str(mseTest)]; title(titlestr);
```

Chapter 7

Genetic Programming Based System Identification in HeuristicLab

7.1 Introduction

The HeuristicLab (HL) is our framework for developing and testing optimization methods, parameters and applying these on a multitude of problems. The development of HL was started in 2002 and has meanwhile lead to to a stable and productive optimization platform; it is continuously enhanced and topic of several publications ([WA04c], [WA04a], [WA04b], [WA05a] and [WWP$^+$07]). On the respective website[1] the interested reader can find information about the design of HeuristicLab, its development over the years, installable software, information, documentation and publications in the context of HeuristicLab and the research group HEAL[2].

One of the most beneficial aspects of HeuristicLab is its plug-in based architecture. In software engineering in general, plug-in based software systems have become very popular; by not only splitting the source code into different modules but compiling these modules into enclosed ready to use software building blocks, the development of a whole application or complex software system is reduced to the task of selecting, combining and distributing the appropriate modules. Due to the support of dynamic location and loading techniques offered in modern application frameworks as for example Java or .NET, the modules do not need to be stati-

[1] `http://www.heuristiclab.com/`
[2] Heuristic and Evolutionary Algorithms Laboratory, Linz / Hagenberg, Austria

cally linked during compilation but can be dynamically loaded at runtime. Thus, the core application can be enriched by adding these building blocks, which are therefore called “plug-ins”.

Problem representations, solution encodings and numerous algorithmic concepts have in the meanwhile been developed for HeuristicLab realizing a large number of heuristic and evolutionary algorithms (GAs, GP, evolution strategies, tabu search, etc.) for a wide range of problem classes including the traveling salesman problem, the vehicle routing problem, real-valued test functions in different dimensions, and, last, but not least, also data based modeling.

But not only the software platform itself is flexible and extensible, also the algorithms created in HL are (since version 2.0) not fixed and can be not only parameterized, but even designed by the user. This is possible by organizing all solution generating and processing methods into operators working on single solutions or sets of solutions. Standardizing this interaction can be done by defining a small set of fixed interfaces; a solution processing operator is simply a method taking a single solution (for example evaluation) or a set of solutions (for example selection) and returning a single solution or multiple solutions.

By providing a set of plug-ins, each realizing a specific solution representation or operation, the process of developing new heuristic algorithms is revolutionized. Algorithms do not need to be programmed anymore but can be created by combining different plug-ins. This approach has a huge advantage: By providing a graphical user interface for combining plug-ins, no programming or software engineering skills are necessary for this process at all. As a consequence, algorithms can be modified, tuned or developed by experts of different fields with less or no knowledge in the field of application development. This transfers development from software engineering to the concrete application domains profiting from the profound knowledge of the users and eliminating the communication overhead between optimization users and software engineers. [WWP+07]

Please note that by doing so, of course data preprocessing steps such as filtering, downsampling and other operations can also be integrated into the whole GP-based identification algorithm. Still, as data preprocessing is again a highly non-trivial task on its own consisting of several critical issues, we are not going to go into this topic any further.

So, this extensible and flexible concept enables us to combine the advanced GA concepts with genetic operators for GP; operators for analyzing dynamics in GP populations can be integrated as well as evaluators that compare training, validation and test qualities.

Here we want to summarize how system identification problems are represented in HeuristicLab and how we have designed an appropriate solution encoding and respective operators.

7.2 Problem Representation

A system identification problem instance has to include all data which are needed in genetic programming for generating models describing the underlying system's behavior.

7.2.1 The Data Base and Data Partitions

The most important part of the representation of a system identification problem, that is to be tackled with genetic programming, is the *data collection* storing all available measurement data; the index of the *target variable* also has to be known and available for the modeling algorithm.

Furthermore, there also has to be an indication which data samples are to be used as *training*, *validation*, and *test* data (in our case given as start and end sample indices). The use of these data segments is different for each particular partition:

- *Training data* are the real basis for the algorithm; the modeling algorithm is allowed to use these training examples of the input / output behavior of the system at hand (or rather of the model that is to be learned) for determining the quality of solution candidates (which in our case here are models / formulae).
- *Validation data* are available for the training algorithm, but normally not used for the essential evolutionary optimization process. These data can for example be used for detecting overfitting phenomena, for pruning or selecting the model that is eventually returned.
- *Test data*, finally, may not be considered by any part of the training algorithm. These data shall be used for testing the models created on new data, i.e. data not included in the algorithm's data base, so that we can determine whether the algorithm was able to generate appropriate models or not.

Additionally, there also has to be a possibility to state which variables of the data base are really available for the modeling algorithm. For example, this becomes relevant when sensor data is included in the data base and used for statistical analysis (correlation analysis, automated fault detection, etc.), but the models that are to be generated for a certain target variable are still not supposed to contain these variables.

7.2.2 Scaling and De-Scaling Basic Problem Data

In theory, variances in the ranges of the variables should be no problem for the GP process simply because multiplicative factors of any range can be produced. Still, in several applications we have seen that approximately equally ranged variables can be advantageous for GP's ability to produce reasonable models.
Thus, a mechanism for scaling and de-scaling variables has been integrated into the problem data management: Variables can be scaled using linear, exponential or logarithmic transformations; the respective parameters are stored as metadata for the basic data so that models created for manipulated data can be re-formulated and applied to original data without changing the model's semantics. Of course, this also allows the application of models, that were originally created for a manipulated data base, to the original data base or any other manipulated variant of the original variables collection.

To be a bit more precise, the scaling functions available are:

- The linear transformation function lt needing two parameters, namely an additive offset o and a multiplicative factor m. Applying this function to a value x returns $lt(o, m, x) = (x + o) * m$. This function is used for linearly scaling data to fixed ranges (as for example [-1,+1] or [0, 100]) or to Gaussian distributed variables with mean average 0 and standard deviation 1 (which is done by subtracting the respective variable's average and dividing by the variable's standard deviation).
  The inverse function to $lt(o, m)$ is $lt^{-1}(o, m)$: $lt^{-1}(o, m, x) = x/m - o$.
- The exponential function exp which needs no parameters: $exp(x) = e^x$. The inverse function to exp, exp^{-1}, is the logarithmic function log.
- The logarithm function log which also needs no parameters: $log(x) = ln(x)$. It is the inverse function to exp, and exp is also the inverse function to log.

Let us assume the following examples: Be T the target variable and X_1, X_2

and X_3 potential input variables; furthermore be m a model created for the given data after scaling T to $T^* = lt_1(o_1, m_1, T)$, X_2 to ${X_2}^* = log(X_2)$ and X_3 to ${X_3}^* = lt_2(o_2, m_2, X_3)$.
If a model which was created on modified data is to be evaluated on the original data base, then all references to modified input variables have to be replaced by functions that introduce the respective transformations into the model, and the inverse transformation of the target variable has to be applied to the whole expression by introducing the respective inverse function into the model. In out example, let m be

$$m : T^* = X_1 + {X_2}^*/{X_3}^* \tag{7.1}$$

and the model m' the respective model equivalent to m with respect to original data. By introducing the transformations of input variables into the model and applying the inverse of the target variable's transformation to the resulting model we get

$$m \quad : \quad T^* = X_1 + log(X_2)/lt_2(o_2, m_2, X_3) \tag{7.2}$$

$$\Leftrightarrow m' \quad : \quad T = \frac{X_1 + log(X_2)/lt_2(o_2, m_2, X_3)}{m_1} - o_1 \tag{7.3}$$

Of course, this procedure can also be applied vice versa, i.e. when it comes to transforming a model created for (or transformed so that it fits to) original data into an equivalent one that is applicable to modified data. In this case, the transformations applied to the input variables have to be inverted and the target variable's transformation applied to the whole resulting expression.

7.2.3 Definition of Minimum and Maximum Time Offsets

Pure availability of a variable is still not sufficient information; what we also need is whether and which time offsets are allowed when referencing a variable. For example, let y be the target variable and u, v and w possible input variables for a model for y; as we want to model y for time (sample) t we search for a model for y_t. The first crucial decision to be made is whether we want to generate static or dynamic models: In static models, only inputs at time t are considered for describing the target variable at time t and the target y_t would be described as a function $f : y_t = f(u_t, v_t, w_t)$. In dynamic modeling, on the contrary, input variables can also be referenced with a certain time lag meaning that not values of time t are used but rather "historic" data; for example, f could then be a function modeling y_t using u_{t-4}, v_{t-1}, v_{t-2} and w_t.

In several application scenarios one also explicitly excludes input values of time t; what we get by excluding contemporary input data is a prediction model that can also be used for modeling future values on the basis of previously measured / recorded data.

Furthermore, the generation of autoregressive models also becomes possible: Autoregressive models are formulas that model an output y_t incorporating previous outputs $y_{t-1}, y_{t-2}, \ldots, y_{t-t_{max}}$; an exemplary autoregressive model for our example could be $f_{AR} : y_t = u_t + v_{t-2} + y_{t-1}$.
So, as the target variable can also be used with certain time offsets, GP is also able to generate autoregressive models.

7.2.4 Metadata

Lots of additional information for system identification problem instances can also be not essentially necessary, but very useful in the modeling process:

- Complexity limits for the models that are to be created can be given as maximum values for the height as well as the size of the models. Height hereby is equal to the height of the respective model structure tree as is to be described in Section 7.4, size refers to the number of nodes of the structure tree.
- Meta-Information such as descriptions of the data and the underlying system or descriptions and names of the variables in the data base, e.g.
- A collection of function and terminal definitions that can be used for compiling and evaluating models – a detailed description about the management of function bases is about to come in Section 7.3.
- The best solutions found so far - this of course also has to include at least information about
  - the data partitions used as training and validation data,
  - the evaluation operator and respective parameter settings applied for evaluating solution candidates,
  - which variables were used in the modeling process applying which minimum and maximum offsets, and
  - the function and terminal definitions that were available for compiling and evaluating models.

7.3 The Functions and Terminals Basis

7.3.1 Motivation, Introduction

The correct design of the functions and terminals basis used for compiling and evaluating formulas is one of the most crucial issues in the design of a GP-based system identification approach; for the sake of simplicity we are in the following going to refer to this pool of definitions of functions and terminals as *functional basis*. In fact, this is not wrong at all, anyway: Terminals can also be seen as functions taking no argument. As we will see in the following, terminal definitions are also functions that take several inputs such as a reference to the data basis, the variable and sample indices, the (time) offset and a concrete coefficient for calculating the returned value. Still, as the handling of terminals differs a lot of the handling of functions, we also treat them separately whenever necessary.

As the HeuristicLab and all plug-ins are implemented in C# using the .NET framework, the most obvious approach would be to use the functions of the .NET framework for building models; essentially, this was done in our GP implementation for the versions 1 and 1.1 of HL ([Win04], [WAW05a], [WAW05b], [WAW06a], [WAW06e], [WAW06c], [WEA+06]).

During own research activities and in the course of discussion with research partners in academics as well as industries we became more and more convinced that it would be a great benefit for GP based modeling if the users were able to program and parameterize the functions and terminals by themselves. So, starting from the implementation in HL 2.0, a flexible and user-programmable functional basis has been used.

The definition of the evaluation of functions and terminals surely is the core of any functions and terminals management unit. So, for each function as well as for every terminal definition we have to be able to manage the source code that represents its evaluation definition, to compile it and provide compiled functions to the GP process.

In detail, these definitions are designed and implemented in HeuristicLab as is explained in following sections.

7.3.2 Definition of the Evaluation of Terminals

The definition of a terminal evaluation is a function that requires a reference to the data basis, variable and sample indices, a sample offset and a coefficient as inputs; depending on the selected terminal definition, this information is processed and the return value calculated. So, a terminal definition t is a function of the data collection D, the variable index v, the sample (time) index s, a sample offset o and a coefficient c.

Let us consider the following examples t_{var}, t_{diff} and t_{const} representing standard variable, differential and constant definitions:

$$\begin{aligned}
t_{var}(D, v, s, o, c) &= c * D[v, s - o] \\
t_{diff}(D, v, s, o, c) &= c * (D[v, s - o] - D[v, s - o - 1]) \\
t_{const}(D, v, s, o, c) &= c
\end{aligned}$$

t_{var} calculates the product of the given coefficient multiplied with the value of variable v at sample index s shifted by o indices, thus taking the value $D[v, s - o]$. t_{diff} calculates the difference of the referenced values at $D[v, s - o]$ and its predecessor, $D[v, s - o - 1]$, and returns it multiplied with the coefficient c. t_{const}, finally, simply returns the given coefficient and thus represents a constant terminal definition.

The definition of such a terminal can of course become arbitrarily simple or complex, depending on the user's intention. Anyway, in HL the definition of the evaluation functions has to be done in C# notation using the following interface:

```
public double TerminalEvaluation(double[][] Data,
    int Var, int Sample, int Offset, double Coeff)
```

The implementation of a terminal definition thus is a method following the interface given above. The respective core method source codes for the exemplary terminals t_{var}, t_{diff} and t_{const} could be defined in the following way:

t_{var} : `return Coeff * Data[Var][Sample-Offset];`

t_{diff} : `return Coeff * ( Data[Var][Sample-Offset] -`
`Data[Var][Sample-Offset-1] ) ;`

t_{const} : `return Coeff;`

7.3.3 Definition of the Evaluation of Functions

The interface for function evaluation definitions is a lot simpler than the evaluation interface for terminals as described above: A function is simply defined by the way how it calculates a value given a set of input values. Additionally, we also use a variant index so that it is possible to define several variants of functions within one function definition. So, a function definition f is a function of the input data vector *input* and the variant v.

Let us consider the following examples f_{add}, f_{div} and t_{trig} representing addition, division and trigonometric functions:

$$\begin{aligned} f_{add}(input, v) &= sum(input) \\ f_{div}(input, v) &= input[1]/input[2] \\ f_{trig}(input, v) &= \begin{cases} \sin(input[1]) & : \; v = 1 \\ \cos(input[1]) & : \; v = 2 \\ \tan(input[1]) & : \; v = 3 \\ error & : \; otherwise \end{cases} \end{aligned}$$

f_{add} calculates the sum of all input values, f_{div} divides the first argument by the second one, and f_{trig} returns the sine, the cosine or the tangent of the first input, depending on the value of the variant index passed.

In HL the definition of the evaluation functions has to be done using the following interface:

```
public double FunctionEvaluation(double[] Args, int Var)
```

The implementation of a function definition thus is a method following the interface given above; the respective core method source codes for the exemplary terminals f_{add}, f_{div} and f_{trig} could be defined in the following way:

f_{add} :
```
double d = 0;
for (int i=0; i<Args.Length; i++)
     d += Args[i];
return d;
```
f_{div} :
```
if(Args[1]==0) return double.NaN;
return (Args[0] / Args[1]);
```
f_{trig} :
```
if (Var==0) return Math.Sin(Args[0]);
if (Var==1) return Math.Cos(Args[0]);
if (Var==2) return Math.Tan(Args[0]);
throw new Exception("Unknown function variant");
```

Of course, logical functions can so be integrated into the functions pool as well as Boolean functions connecting logical and Boolean functions.

Please note that the functions interface definition actually implemented in HL is a bit more sophisticated. In fact, what is also handed over to the function is an array storing a certain number of previously calculated values, i.e. a history of exactly this function:

```
public double FunctionEvaluation(double[] Args, int Var,
    double[] History)
```

If the history array is used in the evaluation function, then a number of predefined calculated values are automatically saved in an appropriate array, stored and given to the function at its next evaluation. Thus, it is for example possible to implement an integral function f_{int} using the history *hist*:

$$f_{int}(input, v, hist) = hist[1] + input[1]$$

which in HL / C# notation could be implemented as

$$f_{int} \; : \; \texttt{return = History[0] + Args[0];}$$

7.3.4 String Representations of Terminals and Functions

Even though the evaluation on a given data basis is the most important task for a model, appropriate string representations are also necessary for representing formulas in prefix notation, standard notation (as a mixture of infix and prefix notations) or in such a way that they can be immediately incorporated in MATLAB©, Mathematica©, LaTeX or C/C++/C# program code.

For each representation variant there are specific interfaces for terminals and functions; in all cases character strings are returned, but the input parameters vary significantly. The functions terminal string representations are given the same parameters as the evaluation functions (except for a reference to the data basis) and, in some cases, the variable name; string representation methods for functions take string respective string representations of the function's inputs and return composed strings representing the function and its inputs.

In the following we here use the standard (infix/prefix) notation for demonstrating the mechanisms that are to be described. For terminals and variables we use the interfaces

```
public string Terminal_Standard(int Var, string VarName,
    int Offset, double Coeff)
```
and
```
public string Function_Standard(string[] Args, int Var),
```

respectively. For standard variables and the addition function, for example, the respective method implementations could be given in the following way:

t_{var} :
```
string s = "[" + Coeff.ToString() + "*";
s = s + VarName;
if (Offset==0) s = s+"(t)";
else s = s+"(t-" + Offset.ToString() + ")";
return (s + ")]");
```
f_{add} :
```
string s = "(" + Args[0];
for(int i=1; i<Args.Length; i++)
     s = s + "+" + Args[i];
s = s + ")";
return s;
```

The standard string representations of two terminals referencing variable w with time-offset 4 and coefficients 1.2 and 0.9, respectively, and their addition would so result in `[1.2*w(t-4)]`, `[0.9*w(t-4)]`, and `([1.2*w(t-4)]+[0.9*w(t-4)])`.

7.3.5 Parameterization of Terminals and Functions

Apart from the definitions of evaluation and string representation of terminals and functions there are several respective parameter settings; these are summarized in this section.

Terminal definitions can be parameterized in the following ways:

- The data type and the distribution function of the coefficients allowed has to be defined: Coefficients can be
  - either integral values or real-valued, and
  - their distribution can be either uniform (defined by minimum and maximum values) or Gaussian (defined by average μ and standard deviation σ).
- The set of possible parent types can be defined, i.e. the user is able to declare which functions are allowed to use the respective terminal as input and which ones are not allowed to do so.
  This selection of possible parent functions can be done either explicitly by

selecting a set of functions that are allowed as direct parents, or implicitly by defining which functions are not allowed as parents of the respective terminal type.

The parameterization possibilities for function definitions are even more than those for terminals:

- An arbitrary number of variants can be defined. Apart from considering these variants in the method code (as can be seen in the code for the trigonometric function definitions in Section 7.3.3), each variant can be activated and deactivated independent of the other variants.
- Additionally, for each variant the function's arity (its number of input parameters) has to be defined. The arity can be either fixed or given as a range defined by minimum and maximum values.
- Each function has to define its neutral element(s), also called identity element(s). In binary operations working on elements of the set X, an element e of X is called left identity with respect to the operation $\circ$ if $e \circ a = a$ for all elements a in X; in analogy to this, e is called right identity with respect to $\circ$ if $a \circ e = a$ for all elements a in X.
  This concept of elements that leaves other elements unchanged when combined with them is here used in a more general way as we define neutral (identity) elements for each possible input index of a function:
  - There can be one neutral element that is used for all input indices, or
  - neutral elements can be defined for each possible input index independently.

  For the addition or subtraction functions, e.g., the neutral element for all possible indices is 0, for the multiplication function it is 1 for all inputs. But when it comes to the division, then the identity elements have to be defined separately for each input: As we divide the first input by the second one, the neutral element for the first index is 0, whereas for the second input it is 1 (because $0/a = 0$ and $a/1 = a$ for all $a \in \mathbb{R}$).
- Similar to the parent type restrictions that can be set for terminals, functions can also define a set of valid parent function definitions. Again, this can be done either directly or indirectly by selecting functions that are not allowed as parent function types.

- Finally, functions can also define child type restrictions. This can also be done directly by selecting certain function or terminal definitions as valid child types (i.e. types that are allowed as inputs for the function), or by explicitly excluding certain types from the set of possible input definitions.
  In order to maximize the flexibility of this child type management concept, these selections can be done either for all input indices uniformly or for each input index separately.

Function and terminal definitions and their respective parameterizations are collected in functional and terminal management units which we here, as already mentioned before, call "functional bases". In each functional basis we not only store function and terminal definitions, but also which ones are activated and which ones are not, and an initial weighting factor is also given for each definition denoting its relative probability to be chosen when it comes to selecting a randomly chosen function or terminal.

7.4 Solution Representation

7.4.1 Representing Formulas by Structure Trees

As we have now described how function and terminal definitions are managed, we shall now take a look at the representation of solution candidates for genetic programming based system identification. The most intuitive way to represent models is modeling them as structure trees; starting with Koza's first GP attempts using LISP structure trees, the concept of trees representing formulas has had a long tradition in GP (see [Koz92], [KKS$^+$03a], [LP02], [Kei02] or [PMR04], e.g.).

Structure trees consist of nodes and references from parent nodes to their children. Thus, for representing formulas we have to create node structures that are able to store all parameters needed as well as references to the function and terminal definitions used; this concept is visualized in Figure 7.1.

The following parameters have to be stored by structure nodes in addition to references to their function or terminal definition:

- Each terminal node has to store the index of the variable it references, the samples (time) offset and the value of the coefficient that is to be used as a multiplicative factor.

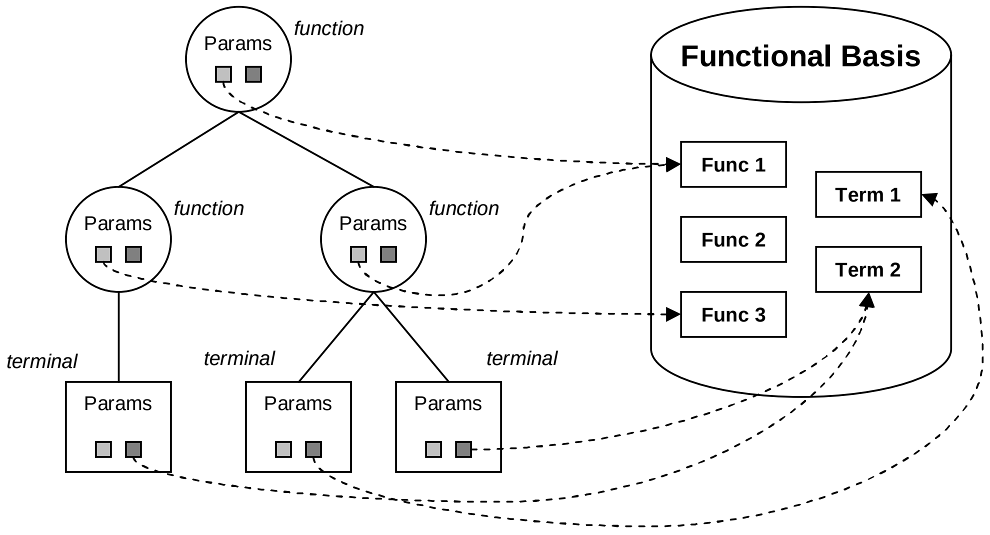


Figure 7.1: Structure tree representation of a formula.

Thus, when it comes to evaluating a terminal node for a given data base and a certain sample index, the referenced terminal definition is called using the given data and sample index as well as the parameters stored in the node; the value returned by the terminal definition function is returned as result of the node's evaluation or representation method.

- A function node has to store not only references to its child nodes and a function definition, but also the index of the function's variant. So, when it comes to evaluating a function node or compiling its string representation for a given data base and a specific sample index, the children nodes are first evaluated with these data and then the referenced function is called with the children's returned values and the variant index stored. The result of this function call is then returned as the result of the node's evaluation. As we have described in Section 7.3.3, some functions also consider previously calculated values. So, function nodes additionally have to manage history arrays in which the calculated values are stored and which are also given to the function definition at the next sample's evaluation.

7.4.2 Operators for Initializing and Manipulating Model Structures

7.4.2.1 Initialization

The initialization of structure trees is essentially the compilation of random tree structures referencing to randomly chosen function and terminal definitions. Of course, all constraints given by the functional basis have to be considered:

- The number of children of each function node has to fulfill the arity constraints given by the function definition parameterization; in the case of fixed arity the number of children has to be exact this value, and in the case of variable arities the number of children may not fall below the minimum or rise above the maximum arity limit.
- Of course, parent and child constraints also have to be considered.
- The structure complexity given in the problem representation (regarding height and size of structure trees) may not be exceeded.
- Variable indices are chosen according to variable availabilities, sample offsets are initialized according to minimum and maximum sample offsets defined by the problem instance.
- Coefficients of terminal nodes are initialized according to parameter settings defined in the terminal definition.

7.4.2.2 Crossover

The most frequently used crossover operator is the single-point subtree exchanging crossover already described in detail in Section 3.2.1.3. Subtrees are exchanged and new formulae are formed, the references to the function and terminal definitions are copied into the new solution candidate, Figure 7.2 illustrates this mechanism.

Of course, all constraints defined by the functional basis have to be satisfied here, too. Especially child and parent relations of the new combinations have to be checked and invalid constellations avoided. The complexity limitation requirements given by the problem instance also have to be fulfilled.
In fact, we have implemented and use three different types of crossover operators:

- The *standard* crossover variant chooses subtrees without considering their size.
- The *low level* crossover variant tries to exchange rather small subtrees of height 1 or 2, e.g.
- The *high level* crossover variant tries to exchange rather big subtrees as for example the roots' children.

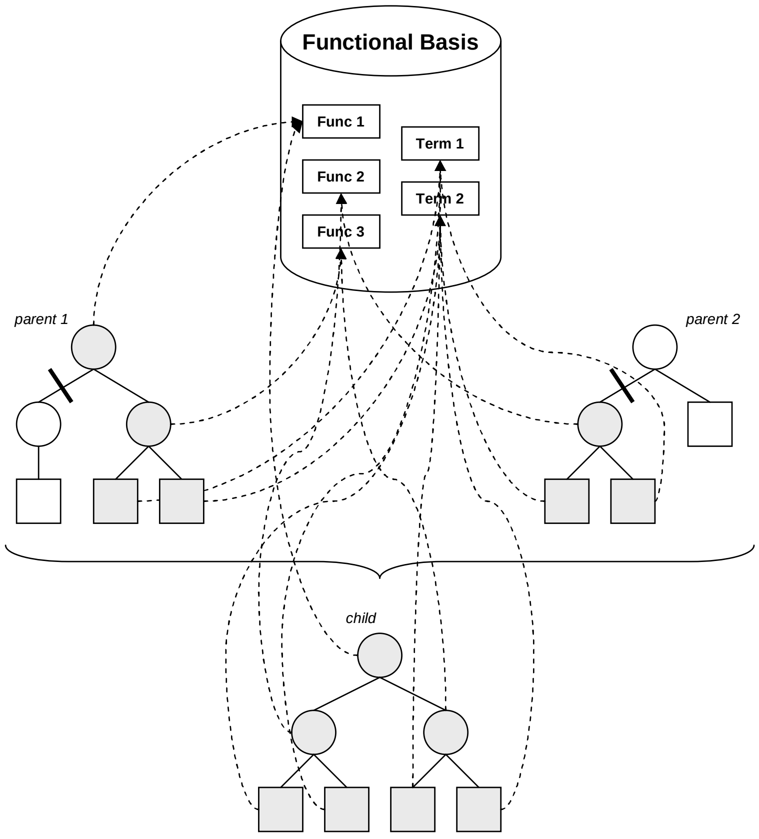


Figure 7.2: Structure tree crossover and the functional basis.

7.4.2.3 Mutation

Finally, mutating a structure tree can be done in several different ways. Some structural as well as parametric mutation variants are as follows:

- A sub-tree could be deleted or replaced by a randomly re-initialized sub-tree.

- A function node could for example change its function type or turn into a terminal node.
- A terminal node representing a variable could for example change its index and thus in the following refer to another variable.
- A terminal node representing a constant could be multiplied with a factor. A good choice for the distribution of these multiplicative mutation factors could be a Gaussian distribution with average 1.0 so that the probability of smaller changes is greater than the probability of larger modification.

Up to now we have always stressed the fact that complexity limitations are given in the problem representation of the concrete system identification problem at hand. In fact, complexity limitations can also be defined by crossover and mutation operators; these operators can be parameterized so that they produce models by crossing parents or mutating formulas that fulfill size or height restrictions independently of the settings given in the problem. These limitations can for example also be modified during the execution of the GP process.

7.5 Solution Evaluation

7.5.1 Standard Solution Evaluation Operators

The primary task of an evaluation operator estimating the fitness of a system identification solution candidate is surely to measure how well the values calculated using the model fit the original target values. Numerous different evaluation functions are possible and have been reported on in the literature; in principle, the estimated values **e** (calculated evaluating the model on the given data basis) are compared to the original target values **o**. In this context it is for the function irrelevant whether the model is evaluated on training, validation, test or any other data partition.
Here we describe three rather simple functions that have also been implemented as evaluation operators for HeuristicLab:

- The *mean squared errors function* (MSE) has in fact already been described; the function returns the average value of the squared residuals of **e** and **o**:

$$MSE(e,o) = \frac{1}{N}\sum_{i=1}^{N}(e_i - o_i)^2; N = |e| = |o| \tag{7.4}$$

- The *coefficient of determination* (R^2) function can be used for measuring the proportion of a variable's variability that is accounted for by the model that tries to explain it; it can also be seen as the ratio of the variability of the modeled target values to the variability of the original target values. R^2 of original and modeled target values, **o** and **e**, respectively, is defined as

$$R^2(e,o) = 1 - \frac{SS_E}{SS_T}; \tag{7.5}$$

$$SS_E = \sum_{i=1}^{N}(o_i - e_i)^2, SS_T = \sum_{i=1}^{N}(o_i - \bar{o})^2, \tag{7.6}$$

$$\bar{o} = \frac{1}{N}\sum_{i=1}^{N} o_i, N = |e| = |o| \tag{7.7}$$

  where SS_E stands for the explained sum of squares and SS_T for the total sum of squares of the original values. The better a model is, the more the R^2 value converges to 1.

- The *variance accounted for* (VAF) function is defined as the fraction of the variances of the residuals and the original target values:

$$VAF(e,o) = 1 - \frac{var(o-e)}{var(o)}; \tag{7.8}$$

$$var(x) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x}), \bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i, N = |x| \tag{7.9}$$

  The variance of the residuals, i.e. the differences between the original and modeled values, is so divided by the original values' variance; the smaller the residuals' variance is, the nearer the calculated value converges towards 1. This main difference of this evaluation function compared to other ones as for example mse or R^2 is that it does not punish constant residuals; only the variance of the residuals is taken into account and might decrease a model's quality.

In the implementations of these evaluation functions we have introduced a parameter for limiting the maximum contribution of a single sample's error to the total evaluation. The residual of each specific sample can so be limited in relation to the original target values' range; this is supposed to help to cope with outliers and invalid values calculated by division by 0, e.g.

7.5.2 Combined Solution Evaluation

Several advanced evaluation concepts are also realized in an advanced evaluation operator for HeuristicLab. Again, for the explanations given in this section let **o** be the original and **e** the estimated target values, and N the number of samples analyzed; furthermore, let $range(o)$ be the range of the original target values:

$$range(o) = max(o) - min(o) \tag{7.10}$$

First, instead of mean squared errors we use the *mean exponentiated error* function; the residuals are raised to the power of n, a parameter of this particular evaluation function, and the mean value of these exponentiated errors is calculated:

$$MEE(o,e,n) = \frac{1}{N}\sum_{i=1}^{N} |e_i - o_i|^n ; N = |e| = |o| \tag{7.11}$$

Additionally, this operator is able to combine the evaluation functions given in the previous section; a combined fitness value is calculated as a linear combination of the three separate fitness values.
First, the fitness values $MEE(o,e,n)$, $R^2(o,e)$ and $VAF(o,e)$ have to be scaled so that they have comparable ranges. The exponentiated errors are scaled by dividing them by a fourth of the target values' range, so for calculating the scaled fitness value $MSEE'(o,e,n)$, $MSE(o,e,n)$ is divided by a fourth of the target data's range raised to the power of n since

$$\begin{aligned}
MEE'(o,e,n) &= \frac{1}{N}\sum_{i=1}^{N}\left(\frac{|e_i - o_i|}{\frac{range(o)}{4}}\right)^n ; N = |e| = |o| && (7.12)\\
MEE'(o,e,n) &= \frac{1}{N}\left(\frac{1}{\frac{range(o)}{4}}\right)^n \sum_{i=1}^{N} |e_i - o_i|^n && (7.13)\\
MEE'(o,e,n) &= MEE(o,e,n) * \left(\frac{1}{\frac{range(o)}{4}}\right)^n && (7.14)
\end{aligned}$$

where n is the exponent chosen for raising the errors to the power of n.
The scaled values $R^{2\prime}(o,e)$ and $VAF'(o,e)$ are calculated as simply as

$$R^{2\prime}(o,e) = 1 - R^2(o,e), VAF'(o,e) = 1 - VAF(o,e) \tag{7.15}$$

since the range of the R^2 and VAF functions is [0,1], anyway.

The minimum and maximum residuals $r_{min}(o,e)$ and $r_{max}(o,e)$ can also be considered; before using them in the combined fitness function, they are scaled in the same way as the MEE values:

$$r = e - o; r_{min}(o,e) = min(r), r_{max}(o,e) = max(r) \tag{7.16}$$

$$r_{min}'(o,e) = \frac{r_{min}(o,e)}{\frac{range(o)}{4}}, r_{max}'(o,e) = \frac{r_{max}(o,e)}{\frac{range(o)}{4}} \tag{7.17}$$

All these scaled partial fitness contribution values are multiplied with coefficients c_1, c_2, c_3, c_4, c_5, summed and the result divided by the sum of coefficients; the result is returned as the combined fitness value $COMB(o,e,n,c)$:

$$\begin{aligned}
a_1 &= c_1 * MEE'(o,e,n) && (7.18)\\
a_2 &= c_2 * R^{2\prime}(o,e,n) && (7.19)\\
a_3 &= c_3 * VAF'(o,e,n) && (7.20)\\
a_4 &= c_4 * r_{min}'(o,e) && (7.21)\\
a_5 &= c_5 * r_{max}'(o,e) && (7.22)\\
COMB(o,e,n,c) &= \frac{a_1 + a_2 + a_3 + a_4 + a_5}{c_1 + c_2 + c_3 + c_4 + c_5} && (7.23)
\end{aligned}$$

There are, in fact, even more sophisticated evaluation operators to be described, namely a time series analysis specific one as well as a classification specific one. These are about to be discussed in Sections 8.1 and 8.2.

7.5.3 Adjusted Solution Evaluation

A modification of the coefficient of determination function R^2 is the so-called *adjusted* R^2; when evaluating a model m, then this extension of the R^2 function described above also takes into account the number of explanatory terms of the model. Let N be the sample size, t the number of terms in m, and o and e again the original and estimated target values, so $R^2{}_{adj}(o,e)$ is calculated as

$$R^2{}_{adj}(o,e) = 1 - (1 - R^2(o,e))\frac{N-1}{N-t-1} \tag{7.24}$$

This add-on[3] increases the calculated quality value only if the addition of a new term to the model improves the model's performance more than what would be expected by chance; unlike R^2 it can even become a negative value.

[3]Of course, calling this modification an "add-on" may sound a bit misleading as it is no additive but rather a multiplicative one. The reader is asked to be so kind as to forgive this slight rhetorical incorrectness.

We have adapted this concept in a slightly modified manner so that it is applicable to the partial R^2 and VAF evaluations of the combined evaluator $COMB$ described in Section 7.5.2. These partial evaluation results can be optionally corrected using the factor $\frac{N-1}{N-s-1}$ where s is the model's size, i.e. the number of nodes of the structure tree solution representing the model which is to be evaluated. So, the adjusted evaluation results ${R^2}_{adj}$ and VAF_{adj} are calculated as

$$\begin{aligned} q &= \frac{N-1}{N-s-1} && (7.25)\\ {R^2}_{adj}(o,e) &= 1-(1-R^2(o,e))*q && (7.26)\\ VAF_{adj}(o,e) &= 1-(1-VAF(o,e))*q && (7.27) \end{aligned}$$

7.5.4 Runtime Consumption Considerations

As we have now described all basic genetic operators for data based system identification using genetic programming, we can try to estimate their relative runtime consumption.

The initialization of structure trees is not just called only once, it is also relatively cheap in terms of runtime consumption. This is because nodes, which are relatively small entities, are created according to the rules and limitations given in the problem instance and the functional basis; the connection between nodes is established by references (pointers) from parent to child nodes.

Crossover and mutation are in our case also very inexpensive with respect to runtime and memory consumption. Nodes and references are copied and parameters are modified; only in case of the creation of invalid structure trees it could happen that repair routines have to be used which could, if implemented in a sub-optimal way, cost significant runtime.

Anyway, it boils down to the fact that most of the runtime of a GP based system identification process is consumed by the evaluation of solution candidates. This is because models have to be evaluated on the training (and maybe also validation) data, i.e. on possibly hundreds or thousands of samples. Collecting these values and then calculating the fitness value can be again relatively cheap (with respect to runtime consumption) when using rather simple evaluation functions as those summarized in the Sections 7.5.1, 7.5.2 and 7.5.3. Still, especially when using more complex functions as for example time series analysis or classification specific ones given in Section 8.1 and 8.2, then this part of the evaluation also might cause noticeable runtime consumption.

In HeuristicLab, for instance, we have measured that even when using a graphical user interface with results display and solution protocolling, more than 99.5 % of the algorithm's runtime are consumed by evaluation operators.

7.5.5 Early Stopping of Model Evaluation

So, what can we do to counteract this problem of high computational costs of GP-based structure identification? The simplest answer would be to decrease the size of the training (and validation) data partitions. Of course this is not a generally applicable way to do this; training data should include as much information as possible in a preferably efficient way - it should be as small as possible, but at the same time also as extensive as necessary.

Sampling, i.e. evaluating the models not on the total training / validation data sets but only on certain selected samples seems to be a better idea: By only evaluating the models for a number of (at best randomly) selected sample indices, the total quality is estimated. This on the one hand surely decreases runtime consumption and on the other hand also might help to avoid overfitting as the models are evaluated on different samples at each evaluation step (so that they cannot be fit too closely to a set of samples). Still, the quality measurement might so become somehow instable; a model might be assigned completely different quality values each time it is evaluated because the samples chosen are likely to differ.

When using offspring selection as described in Chapter 4.2 there is even a possibility how to speed up the evaluation without decreasing the quality of the fitness estimation method:

During the offspring selection phase, solution candidates are compared to their parents, i.e. their quality values are compared to their parents' fitness values. In the case of applying most restrictive settings, i.e. when the success ratio is set to 1.0, then models are inserted into the next generation's population only if they fulfill the quality requirements given by the parent's quality values and the comparison factor; there is no pool of possible lucky losers, solution candidates that do not fulfill the given fitness criterion are discarded. In this case the evaluation of a model can be aborted as soon as it is clear that the fitness value will surely not satisfy the fitness criterion even if the rest of the evaluation produces no additional errors.

The issue, then, is how to detect when the evaluation of a model can be aborted without decreasing the quality estimation's accuracy with respect to the total GP process. We introduce a *relative calculation interval size* (*rcis*) which is a value in the interval [0,1] (normally a value as for example 0.1, 0.2 or 0.5) used in the following way:

Let m be a model which is to be evaluated for a system identification problem p; furthermore let $p1$ and $p2$ be the parents of m, and q_{p1} and q_{p2} their respective quality values. The given comparison factor cf is then used for calculating the comparison value cv depending on whether p is a maximization or a minimization problem:

$$q_{min} = min(q_{p1}, q_{p2}); q_{max} = max(q_{p1}, q_{p2}) \tag{7.28}$$

$$q_{range} = |q_{p1} - q_{p2}| \tag{7.29}$$

$$cv = \begin{cases} q_{min} + q_{range} * cf & : \ isMaximizationProblem(p) \\ q_{max} - q_{range} * cf & : \ isMinimizationProblem(p) \end{cases} \tag{7.30}$$

In system identification we normally deal with minimization problems when using the MSE, MEE or $COMB$ evaluation operator as smaller fitness values are favored; when using the R^2 or VAF operator, p can be considered a maximization model since better models are assigned higher fitness values.

Let us now assume that N samples are to be evaluated; $\mathbf{o}$ then is array storing the N original target values, and the calculation samples interval csi is calculated as

$$csi = N * rcis \tag{7.31}$$

The vector of estimated target values $\mathbf{e}$ is initialized as a copy of $\mathbf{o}$; the model's quality q_m is initially set to the worst possible fitness value (-$maxVal$ for maximization, $maxVal$ for minimization problems), and the indices i_1 and i_2 are set to 1[4].

As long as q_m is "better" than cv (i.e., smaller if p is a minimization and greater if p is a maximization problem), the following evaluation steps are executed:

1. The index i_2 is set to $i_1 + csi - 1$; if $i_2 > N$, then $i_2 := N$.
2. The estimated values e_j are calculated for $j = [i \ldots i_2]$; these replace the values at the respective indices in $\mathbf{e}$ so that

$$\mathbf{e} = [e_1, \ldots, e_{i_2}, o_{i_2+1}, \ldots, o_N] \tag{7.32}$$

[4]In this description we again use one-based indexing; in most modern programming languages as C, C++, Java or C#, zero-based indexing would be used instead.

3. q_m is calculated using the given fitness function f:

$$q_m = f(\mathbf{o}, \mathbf{e}) \tag{7.33}$$

4. Now there are several ways how the evaluation is continued:

   (a) If i_2 is equal to N, i.e. if all samples have been considered, then m is assigned the fitness value q_m.

   (b) Otherwise, if q_m is no more "better" than cv, then the evaluation of m can be aborted and m can be assigned the worst possible fitness value ($maxVal$ for maximization, $-maxVal$ for minimization problems).
   As an alternative, we can also assign m an extrapolated fitness value: If p is a minimization problem and the optimal possible fitness value 0, as it is the case if we use the MSE, MEE or $COMB$ operator, then we can assign m the extrapolated fitness value $q_m * \frac{N}{i_2}$.

   (c) Otherwise, go back to step 1 and continue the evaluation of m.

By rearranging the evaluation as described above we guarantee that the quality of models that fulfill the given offspring selection criterion is accurate and calculated in the same way as when using the standard procedure. For models that perform worse than demanded and are therefore not about to fulfill the offspring selection criterion, the evaluation is aborted as soon as it is clear that the evaluation will result in such a "bad" fitness value.
Thus, a lot of runtime can be saved. For the sake of completeness we of course have to admit that the runtime consumption is increased slightly for models that are evaluated on all samples since intermediate fitness values are calculated; still, this minor drawback is accepted since the advantages outweigh it by far.

Chapter 8

Application Domains of Data Based Structure Identification

8.1 Regression, Time Series Analysis and the Design of Virtual Sensors

8.1.1 Regression

In general, modeling numerical data consisting of one (or more) target variables (also called dependent or response variables) using one or more independent variables (also often referred to as explanatory or input variables) is called regression analysis; dependent variables are modeled as functions of the independent variables, corresponding parameters (constants) and an error (noise) term.

This concept of mathematical regression has in principle already been discussed in Chapter 6: What we want to get is a function f that models a target variable t using explanatory variables x and a set of coefficients w such that

$$t = f(x, w) + \epsilon \tag{8.1}$$

where ϵ represents the error (noise) term.

In other words, the main principle can be formulated in the following way: For a given target variable T storing the values $T_{(1)}, \ldots, T_{(n)}$ and a given set of variables $X_1, \ldots, X_N$ we search for a model that describes T as

$$T_{(t)} = f(X_{1(t)}, \ldots, X_{N(t)}) + \epsilon_t \tag{8.2}$$

where ϵ_t is an error term.

Applying this procedure we assume that a model can be created with which it will also be possible to predict correct outputs for other data examples (test sample); from the training data we want to generalize to situations not known (or allowed to analyze) during the training phase. Detailed discussions of theoretical and practical aspects of regression analysis can be found in [Här90], [Fox97] and [DS98], for example.

8.1.2 Time Series Analysis

Whenever (input or output) data of any kind of system are recorded over time and compiled in data collections as sequences of data points, then these sequences are called *time series*; typically, these data points are recorded at time intervals which are often, but not always uniform. The collection of methods and approaches which are used for trying to understand the underlying mechanisms that are documented in time series is called *time series analysis*; but not only do we want to know what produced the data, but what we are also interested in is to predict future values, i.e. we want to develop models that can be used as predictors for the system at hand.

There is a lot of literature on theory and different approaches to time series analysis. One of the most famous approaches is the so-called Box-Jenkins approach as described in [BJ76] and [And76], e.g., which includes separate model identification, parameter estimation and model checking steps. Detailed discussions of other methods and their mathematical and statistic background can be found for example in [And71], [Ken73], [Pan83], [KO90], [Pan91], [BD91], [Ham94] and [BD96]; more recent research and applications are for example given in [PTT01], [Cha01], [Dei04], [Wei06] and [MJK07].

The main principle can be formulated in the following way: For a given target time series T storing the values $T_{(1)}, \ldots, T_{(n)}$ and a given set of variables $X_1, \ldots, X_N$ we search for a model that describes T as

$$\begin{aligned} T_{(t)} = f(X_{1(t)}, X_{1(t-1)}, \ldots, X_{1(t-t_{max})}, \\ \ldots, \\ X_{N(t)}, X_{N(t-1)}, \ldots, X_{N(t-t_{max})}) \quad +\epsilon_t \end{aligned}$$

where t_{max} is the maximum number of past values, and ϵ_t is an error term. If the target variable's values are also allowed to be considered, then a so-called autoregressive part is added so that we search for a model so that

$$\begin{aligned} T_{(t)} = f(X_{1(t)}, X_{1(t-1)}, \ldots, X_{1(t-t_{max})}, & \\ \ldots, & \\ X_{N(t)}, X_{N(t-1)}, \ldots, X_{N(t-t_{max})}, & \\ T_{(t-1)}, \ldots, T_{(t-t_{max})} & \quad) + \epsilon_t \end{aligned}$$

The field of applications of time series analysis (as well as of regression, of course) is huge and includes for example astronomy, sociology, economics or the analysis of physical systems. Of course it is not at all natural that any physical system, may it be technical or not, can be represented by a simple and easily understandable model. In this context the author strongly recommends reading Eugene P. Wigner's article "The Unreasonable Effectiveness of Mathematics in the Natural Sciences" [Wig60]. In this article Wigner points out that, although so many natural phenomena such as e.g. gravitation or planetary motion can be described by astoundingly simple equations, it is not at all natural that "laws of nature" exist and even much less that man is able to discover them.

Especially in the context of analyzing physical systems, the models which are to be created for describing a system can be seen as so-called *virtual sensors*: The goal is to develop models of sufficient quality so that these models (functions) can be used instead of real sensors, i.e. they are virtual sensors. Of course, these virtual sensors can be used in various ways, for example also in addition to real sensors enabling fault detection.

8.1.3 Time Series Specific Evaluation

In this section we shall concentrate on time series analysis with genetic programming: GP is used for evolving models that describe target time series using other data time series collections. Of course we in principle use the GP methods for structure identification described in the previous sections, but some time series specific details are to be described here, especially a time series specific evaluation operator.

In principle there is no reason why one should not use mean squared errors or any other of the evaluation functions already presented for evaluating time series models produced by GP. Still, in time series we do not only want to produce models that approximate the given target values, but also the dynamics of the underlying system that are represented in the measured data. Thus, we also want to estimate a model's quality with respect to the local changes in the data as well as the accumulated values.

This can be done by calculating the differential and integral values. For a given time series $\mathbf{x}$, the differential of order o is defined as $diff(\mathbf{x}, o)$ and the integral as $int(\mathbf{x})$:

$$diff(\mathbf{x}, o)_i = \mathbf{x}_i - \mathbf{x}_{i-o} \tag{8.3}$$

$$int(\mathbf{x})_i = \textstyle\sum_{i=1}^{i} \mathbf{x}_i \tag{8.4}$$

for each index $i \in [1; |\mathbf{x}|]$.

For evaluating a time series model m on the basis of target values $\mathbf{o}$ we calculate all respective values $\mathbf{e}$ by evaluating m and then calculate the combined fitness values (as described in Section 7.5.2) for the plain values, the differential (of a predefined order o) and the integral values. These partial results are weighted using the coefficients c_1, c_2 and c_3, and the final result in the following way:

$$TS(\mathbf{o}, \mathbf{e}, o, n, \mathbf{c_{plain}}, \mathbf{c_{diff}}, \mathbf{c_{int}}, c_1, c_2, c_3):$$

$$a_1 = COMB(\mathbf{o}, \mathbf{e}, n, \mathbf{c_{plain}}) \tag{8.5}$$

$$a_2 = COMB(diff(\mathbf{o}, o), diff(\mathbf{e}, o), n, \mathbf{c_{diff}}) \tag{8.6}$$

$$a_3 = COMB(int(\mathbf{o}), int(\mathbf{e}), n, \mathbf{c_{int}}) \tag{8.7}$$

$$TS(\mathbf{o}, \mathbf{e}, o, n, \mathbf{c_{plain}}, \mathbf{c_{diff}}, \mathbf{c_{int}}, c_1, c_2, c_3) = \frac{\sum_{i=1}^{3} a_i \cdot c_i}{\sum_{i=1}^{3} c_i} \tag{8.8}$$

with $\mathbf{c_{plain}}$, $\mathbf{c_{diff}}$ and $\mathbf{c_{int}}$ being the coefficients needed by the combined evaluation function for weighting the partial MEE, VAF and R^2 results as well as the maximum negative and positive errors.

Of course, early stopping of model evaluations as described in Section 7.5.5 is also possible for this time series evaluation function.

8.2 Classification

8.2.1 Introduction

Classification is understood as the act of placing an object into a set of categories, based on the object's properties. Objects are classified according to an (in most cases hierarchical) classification scheme also called taxonomy. Amongst many other possible applications, examples of taxonomic classification are biological classification (the act of categorizing and grouping living species of organisms), medical classification and security classification (where it is often necessary to classify objects or persons for deciding whether a problem might arise from the present situation or not). A statistical classification algorithm is supposed to take feature representations of objects and map them to a special, predefined classification label. Such classification algorithms are designed to learn (i.e. to approximate the behavior of) a function which maps a vector of object features into one of several classes; this is done by analyzing a set of input-output examples ("training samples") of the function. Since statistical classification algorithms are supposed to "learn" such functions, we are dealing with a specific area of *machine learning* and, more generally, *artificial intelligence*.

In a more formal way, the classification problem can be formulated in the following way: Let the data consist of a set of samples, each containing k feature values $x_{i1}, \ldots, x_{ik}$ and a class value y_i. So, what we look for is a function f that maps a sample x_i to one of the c classes available:

$$f : X \rightarrow C; \tag{8.9}$$

$$\forall (x \in X) : f(x) = f(x_1, \ldots, x_k) = y; y \in \{C_1, \ldots, C_c\} \tag{8.10}$$

where X denotes the feature vector space and C the set of classes.

There are several approaches which are nowadays used for solving data mining and, more specifically, classification problems. The most common ones are (as for example described in [Mit00]) decision tree learning, instance-based learning, inductive logic programming (such as Prolog, e.g.) and reinforcement learning.

8.2.2 Real-Valued Classification Using Genetic Programming

In this section we shall concentrate on GP based classification. In fact, we here restrict ourselves to real-valued classification tasks, i.e.

$$X \subset \mathbb{R}^k, C \subset \mathbb{R} \tag{8.11}$$

Thus, we can apply the GP based system identification approach described in the previous sections; especially the representations of the problems, the solution candidates and the genetic operators can be used without any restrictions.

The only critical aspect is that the evaluation and the quality estimation of classifiers have to be modified: Evaluating a model m on a set of input features $(x_1, \dots, x_k)$ will lead to a target value $y \in \mathbb{R}$, but y does not necessarily have to be exactly one certain class value, i.e. we might get $y \notin C$. The exact mapping of feature vectors and their respective target values to class values is done using sets of thresholds $t1, \dots, t_{c-1}$ placed between the class values $C_1, \dots, C_c$:

$$\forall (i \in [1; c-1]) : C_i < t_i < C_{i+1} \tag{8.12}$$

Based on a set of thresholds T we can classify a sample for which the target value y has been calculated as belonging to class c_t using the mapping function f':

$$\begin{aligned} f' : \{\mathbb{R}, \mathbb{R}\} \rightarrow C \quad &; \quad c_t = f'(c_t, T) & (8.13) \\ y < t_1 \quad &\Rightarrow \quad f'(c_t, T) = C_1 & (8.14) \\ y > t_{c-1} \quad &\Rightarrow \quad f'(c_t, T) = C_c & (8.15) \\ \forall (i \in [1; c-2]) : t_i < y < t_{i+1} \quad &\Rightarrow \quad f'(c_t, T) = C_{i+1} & (8.16) \end{aligned}$$

Figure 8.1 exemplarily shows a graphical representation of the evaluation of classification target data compared to the estimated values calculated. In this case, 3 target values are given ("1", "2", and "3"), and each sample is assigned a class value; for each sample an estimated value is calculated using the trained model (in this case, a formula was trained using GP) and by applying thresholds we can classify each sample as belonging to one of the given classes. The chart is later again shown as Figure 15.2; please see Section 15 for background information about the data set and the GP strategy that has been used here.

8.2.3 Analyzing Classifiers

8.2.3.1 Classification Rates and Confusion Matrices

When it comes to analyzing classifiers, the most important aspect is of course how many samples are classified correctly. For each feature vector sample x we have an original classification y, and by applying the classifier which is to be evaluated we get the predicted class y'. As described before, this classification of x is done using a classification model yielding $y = f(x)$ and an optional post-processing step using thresholds T yielding $y = f'(y, T)$.

Let us assume that we analyze n samples $x_{1\dots n}$ (classified into c classes $C_1 \dots C_c$) with their respective original classifications $y_{1\dots n}$; by applying a classification model

m we get the respective predicted classifications $y'_{1...n}$ as described above. The ratios of correctly classified samples for all classes or each class separately is calculated as cc and cc_i, respectively:

$$cc = \frac{|j : j \in [1; n] \ \& \ y_j = y'_j|}{n} \tag{8.17}$$

$$\forall (i \in [1; c]) : cc_i = \frac{|j : j \in [1; n] \ \& \ y_j = y'_j \ \& \ y_j = C_i|}{|j : j \in [1; n] \ \& \ y_j = C_i|} \tag{8.18}$$

For more detailed analysis, confusion matrices [KP98] contain information about actual and predicted classifications done by classification systems. In general, a confusion matrix cm is a table containing $c \times c$ cells that states how many samples of each given class are classified as belonging to a specific class; for example, each column of the matrix can represent the instances of a predicted class while each row

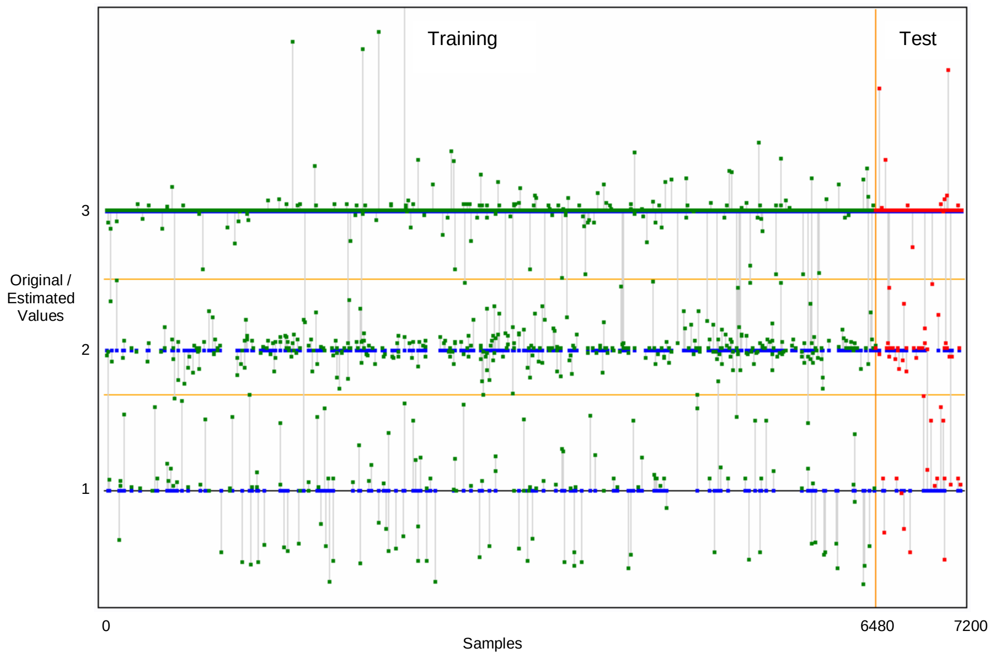


Figure 8.1: Classification example: Graphical representation of a result obtained for the *Thyroid* data set, comparison of original and estimated class values. Original values are drawn as blue spots, estimated values as green spots on training data and red ones for the test data partition; optimal class thresholds (calculated based on the training data performance) are depicted as horizontal yellow lines.

represents the instances in the original (actual) class (or vice versa). So, the value $cm_{i,j}$ stores the number of samples of class i that are classified as class j.
An example is given in Table 8.1 in which each row of the matrix represents the instances in a predicted class while each column represents the instances in the original (actual) class; additionally, the numbers of samples not classified ($nc_1 \dots nc_c$) are also given as well as the total rate of correct classifications. Please note that the sum of all cells has to be equal to the number of samples n, i.e.

$$\sum_{i=1}^{c}\sum_{j=1}^{c} cm_{i,j} + \sum_{i=1}^{c} nc_i = n \tag{8.19}$$

Table 8.1: Exemplary confusion matrix with three classes

		Actual Class			
		"1"	"2"	"3"	
Estimated Class	"1"	$cm_{1,1}$	$cm_{2,1}$	$cm_{3,1}$	
	"2"	$cm_{1,2}$	$cm_{2,2}$	$cm_{3,2}$	
	"3"	$cm_{1,3}$	$cm_{2,3}$	$cm_{3,3}$	
Not classified		nc_1	nc_2	nc_3	
Correct Classifications Ratio					$\frac{\sum_{i=1}^{c} cm_{i,i}}{n}$

The special case of binary classification into two classes (i.e., $c = 2$) is frequently found as it is in many applications necessary to decide for given samples whether or not some given condition is fulfilled. There are the four different possible outcomes of a single predicted (estimated) classification in the case of binary classification into classes "positive" ("yes", "1", "true") and "negative" ("no", "0", "false"):

- A false positive classification is done when a sample is incorrectly classified as "positive" which is in fact "negative",
- a false negative classification is done when a sample is incorrectly classified as "negative" which is in fact "positive", and
- true positive as well as true negative classifications are respective correct classifications.

A typical "positive / negative" example is given in Table 8.2:

In this case,

Table 8.2: Exemplary confusion matrix with two classes

		Actual Class	
		Positive	Negative
Estimated Class	Positive	*a (true positive)*	*b (false positive)*
	Negative	*c (false negative)*	*d (true negative)*

- the *accuracy* is defined as $ACC = \frac{a+d}{a+b+c+d}$,
- the *true positive rate* (also called *sensitivity)* as $TP = \frac{a}{a+c}$,
- the *true negative rate* (also called *specificity)* as $TN = \frac{d}{b+d}$,
- the *false positive rate* as $FP = \frac{b}{b+d}$ (which is in fact the probability of classifying a sample as "positive" when it is actually "negative"),
- the *false negative rate* as $FN = \frac{c}{a+c}$ (which is in fact the probability of classifying a sample as "negative" when it is actually "positive"), and finally
- the *precision* as $P = \frac{a}{a+b}$.

8.2.3.2 Receiver Operating Characteristic (ROC) Curves

Receiver Operating Characteristic (ROC) analysis provides a convenient graphical display of the trade-off between true and false positive classification rates for two class problems [FE05]. Since its introduction in the medical and signal processing literatures ([HM82], [ZC93]), ROC analysis has become a prominent method for selecting an operating point; for a recent snapshot of applications and methodologies see [FBF$^+$03] and [HOFLF04]. ROC analysis often includes the calculation of the area under the ROC curve (AUC).

In the context of two class classification, ROC curves are calculated in the following way: For each possible threshold value discriminating two given classes (e.g., 0 and 1, "true" and "false" or "positive" and "negative"), the numbers of true and false classifications for one of the classes are calculated. For example, if the two classes "true" and "false" are to be discriminated using a given classifier, a fixed set of equidistant thresholds is tested and the true positives (TP) and the false positives (FP) are counted for each of them. Each pair of TP and FP values produces a point of the ROC curve; examples are graphically shown in Figure 8.2. Slightly different versions are also often used, for example the positive predictive value (= TP / (TP

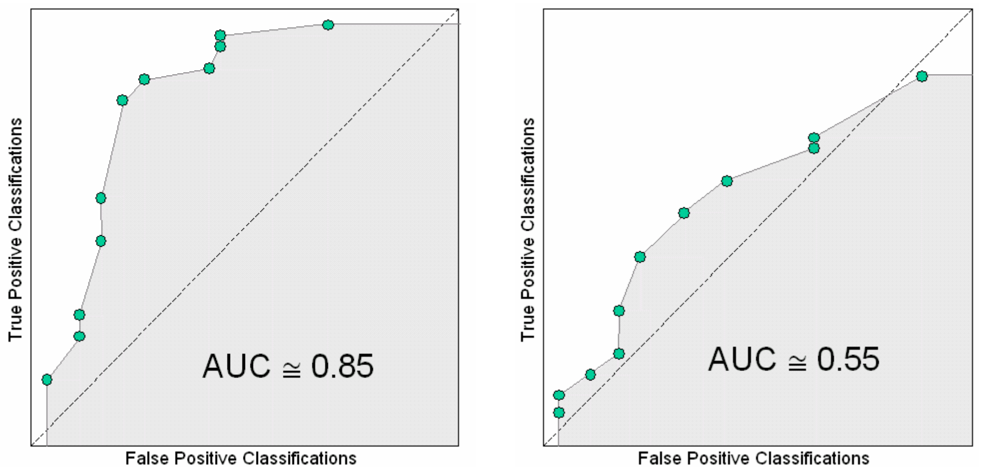


Figure 8.2: Two exemplary ROC curves and the respective areas under these curves (AUCs).

+ FP)) or the negative predictive value (= TN / (FN + TN)) could be displayed instead.

The most common quantitative index describing a ROC curve is the area under it. The bigger the area under a ROC curve is, the better the discriminator model is; if the two classes can be ideally separated, the ROC curve goes through the upper left corner and thus, the area under it reaches its maximal possible value which is exactly 1.0.

This method is very useful for analyzing the quality of two class classifiers, but unfortunately it is not directly applicable for more than two classes. When it comes to measuring or graphically illustrating the quality of multi-class classifiers, one possibility is to define symmetric areas around the original class values; for each class values C_i the corresponding area is defined as $[C_i - r, C_i + r]$. Successively increasing the parameter value r from 0 to $\frac{C_{i+1}-C_i}{2}$ and calculating the numbers of correct and incorrect classifications for each r yields a set of pairs of FP/TP values. Jiang and Motai [JM05], for example, use this technique for illustrating and analyzing the classification performance in the context of automatic motion learning.

Although this method can be used very easily, it is not generally applicable because it is restricted to symmetric areas. Emerson and Fieldsend [FE05] propose a different approach and define the ROC surface for the Q-class problem in terms of a multi-objective optimization problem in which the goal is to simultaneously minimize misclassification rates when the misclassification costs and parameters governing the classifier's behavior are unknown. The problem with this approach is that the estimated Pareto fronts presented in [FE05] can be illustrated and used for graphical interpretation for classification problem involving not more than three classes. This is why we here in the following section propose the use of sets of ROC curves for each class separately.

8.2.3.3 Sets of Receiver Operating Characteristic Curves and their Use in the Evaluation of Multi-Class Classification

In this section we present an extension to ROC analysis making it possible to measure the quality of classifiers for multi-class problems. Unlike other multi-class-ROC approaches which have been presented more or less recently (see [FE05] or [Sri99], e.g.) we propose a method based on the theory of ROC curves that creates sets of ROC curves for each class that can be analyzed separately or in combination. Thus, what we get is a convenient graphical display of the trade-off between true and false classifications for multi-class problems. We have developed a generalization of this AUC analysis for multi-class problems which gives the operator the possibility to see not only how accurately, but also how clearly classes can be separated from each other.

The main idea presented here is that for each given class C_i the numbers of true and false classifications are calculated for each possible pair of threshold between the classes C_{i-1} and C_i as well as between C_i and C_{i+1}. This is in fact done under the assumption that the c classes are ordered and that $C_i < C_{i+1}$ holds for every $i \in [1,(n-1)]$ (with c being the number of classes).

For a given class C_i the corresponding TP and FP values (on the basis of the N original values $\mathbf{o}$ and estimated values $\mathbf{e}$) are calculated as:

$$\forall(\langle t_a, t_b\rangle | (C_{i-1} < t_a < C_i) \,\&\, (C_i < t_b < c_{i+1})) : \tag{8.20}$$

$$TP(t_a,t_b) = |\{e_j : (t_a < e_j < t_b) \,\&\, (t_a < o_j < t_b)\}| \tag{8.21}$$

$$FP(t_a,t_b) = |\{e_j : (t_a < e_j < t_b) \,\&\, (o_j < t_a \vee o_j > t_b)\}| \tag{8.22}$$

This approach has been published first in [WAW06d] and then described in detail (including application examples) in [WAW07a].

The resulting tuples of (FP,TP) values are stored in a matrix which can be plotted as is exemplarily illustrated in Figure 8.3: On the basis of synthetic data

$10^2 = 100$ ROC points for 10 thresholds between the chosen class C_i and C_{i-1} as well as between C_i and C_{i+1} were calculated. This obviously yields a set of points which can be interpreted analog to the interpretation of "normal" ROC curves: the closer the points are located to the left upper corner, the higher is the quality of the classifier at hand.

For getting sets of ROC curves instead of ROC points, the following change is introduced: An arbitrary threshold t_a between the classes C_{i-1} and C_i is fixed and the FP and TP values for all possible thresholds t_b between C_i and C_{i+1} are calculated. What we get is one single ROC curve; this calculation is executed for all possible values of t_a (i.e., for all possible threshold between C_{i-1} and C_i). This procedure also has to be executed the other way around, i.e. also has to choose an arbitrary threshold t_b between C_i and C_{i+1}, calculate all corresponding ROC points and repeat this for all values for all possible values of t_a.

Finally, what we get is a set of ROC curves; an example showing 10 ROC curves is given in Figure 8.3.

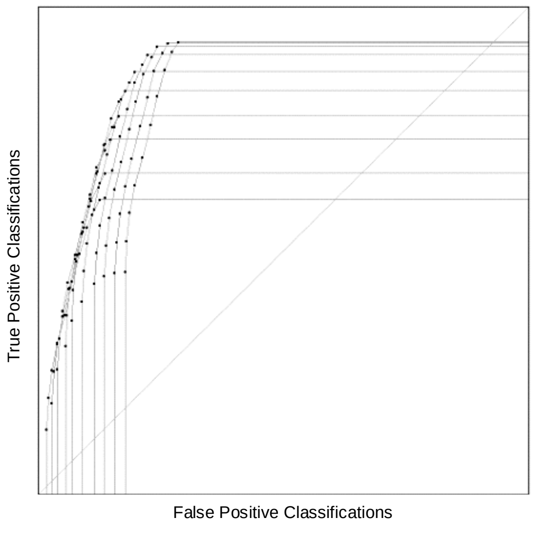


Figure 8.3: An exemplary graphical display of a multi-class ROC (MROC) matrix.

Of course this procedure cannot be executed in exactly this way for the classes C_1 and C_n. For c_1 it is only possible to calculate the ROC points (and therefore the ROC curve) for all possible thresholds between C_1 and C_2, for C_c this is done analogically with all possible thresholds between C_{c-1} and C_c. This is why sets of ROC curves can be calculated for the classes $C_2 \ldots C_{c-1}$ whereas only simple ROC curves can be produced for C_1 and C_c.

As already mentioned in the previous section, the area under the ROC curve (AUC) is a very common quantitative index describing the classifier's quality. In the context of multi-class ROC (MROC) curves the two following values can be calculated assuming that all m ROC curves for a given class have already been calculated:

- The maximum AUC ($MaxAUC$) is the maximum of all areas under the ROC curves calculated for a specific class. It measures how exactly this class is separated from the others using the best thresholds parameter setting.

$$MaxAUC = \max_{i=1..m} AUC(ROC_i)$$

- The average AUC ($AvgAUC$) is calculated as the mean value of all areas under the ROC curves for a specific class. It measures how clearly this class is separated from the others since it takes into account all possible thresholds parameter settings.

$$AvgAUC = \frac{\sum_{i=1..m} AUC(ROC_i)}{m}$$

Thus, what we get is a simple, but surely very useful and intuitive approach extending ROC analysis so that it can be used also in the context of multi-class classification. In addition to a graphical display, the average as well as the maximum area under the resulting ROC curves can be considered for evaluating multi-class classifiers.

In the following we shall see how this can be used in the evaluation of classifiers evolved by GP.

8.2.4 Classification Specific Evaluation in GP

Of course, there is on the one hand no reason why standard evaluation functions such as the MSE / MEE, VAF or R^2 functions could not be used for estimating the quality of classification model during the GP process. The reason for this is that we here, similar to when dealing with regression or time series analysis, want the identification algorithm to produce a model that is able to reproduce the given target data as well as possible.

Still, on the other hand the evaluation of classification models may also include several aspects for which the standard evaluation functions are not suitable. This is why we shall describe several aspects that may contribute to a classification specific evaluation function for GP solution candidates in the context of real-valued learning of classifiers with genetic programming.

8.2.4.1 Preprocessing of Estimated Target Values

Before we compare original and estimated class values we suggest the following classification specific preprocessing step:

The errors of predicted values that are lower than the lowest class value or greater than the greatest class value should not have a quadratic or even worse, but rather partially only linear contribution to the fitness of a model. To be a bit more precise: Given n samples with original classifications o_i divided into c classes $C_1, ..., C_c$ (with C_1 being the lowest and C_c the greatest class value), the so preprocessed estimated values $preproc(e_i)$ shall be calculated as follows:

$$\begin{aligned} &\forall(i \in [1,n]) : \\ &\quad (e_i < C_1) \;\Rightarrow\; preproc(e_i, x) = C_1 - (C_1 - e_i)^{\frac{1}{x}} && (8.23) \\ &\quad (e_i > C_c) \;\Rightarrow\; preproc(e_i, x) = C_c + (e_i - C_c)^{\frac{1}{x}} && (8.24) \end{aligned}$$

with x being an exponential parameter which depends on the evaluation function that uses these preprocessed values. For example, when using the mean squared error or any other function that incorporates the use of squared differences between original and estimated value, x is to be set to 2, whereas when using the MEE function it has to be set to the chosen exponent.

The reason for this is that values that are greater than the greatest class value or below the lowest value are anyway classified as belonging to the class having the greatest or the lowest class number, respectively; using a standard evaluation function without preprocessing of the estimated values would punish a formula producing such values more than necessary.

8.2.4.2 Considering Standard Evaluation Functions

For quantifying the quality of classifiers we can use all functions described in Section 7.5; in contrast to standard applications, we can also apply these functions for each class individually.

In the standard case, all n values are evaluated using the MEE, VAF and R^2 values as well as the minimum and maximum errors $error_{min}$ and $error_{max}$; these can optionally be calculated using the preprocessed values $preproc(e_i)$ instead of e_i for all $i \in [1; n]$. Thus, we get partial values mee, vaf and r^2, $error_{min}$ and $error_{max}$ which can be weighted using the factors w_{mee}, w_{vaf}, w_{r^2}, $w_{err_{min}}$ and $w_{err_{max}}$.

This approach of course does not consider the distribution of samples to the classes; for example, if 98% of the samples belong to class 0 and only 2% to class 1, then the evaluation of a model classifying all samples as 0 will be fairly good when using these standard evaluation functions even though this classifier is more or less useless.

In order to overcome this problem we could for example sample the data so that all classes are represented by the same number of samples; we instead here describe the application of these evaluation functions to the classes given separately:

The sets of estimated values $ec_1 \ldots ec_c$ contain the values estimated for each class $C_1 \ldots C_c$, and in analogy to this the sets $oc_1 \ldots oc_c$ are sets of the corresponding class values:

$$\forall (i \in [1; n]) : o_i = k \Rightarrow e_i \in ec_k, o_i \in oc_k \tag{8.25}$$

Additionally, we also need class weights $w_1 \ldots w_c$ (with $w = \sum_{i=1}^{c} w_i$) and can so calculate the partial fitness values as

$$\begin{aligned}
mee &= \frac{1}{w} \sum_{i=1}^{c} mee(oc_i, ec_i, n) \cdot w_i && (8.26)\\
r^2 &= \frac{1}{w} \sum_{i=1}^{c} r^2(oc_i, ec_i) \cdot w_i && (8.27)\\
vaf &= \frac{1}{w} \sum_{i=1}^{c} \left(1 - \frac{var(oc_i - ec_i)}{var(o)}\right) \cdot w_i && (8.28)\\
error_{min} &= \frac{1}{w} \sum_{i=1}^{c} r_{min}(oc_i, ec_i) \cdot w_i && (8.29)\\
error_{max} &= \frac{1}{w} \sum_{i=1}^{c} r_{max}(oc_i, ec_i) \cdot w_i && (8.30)
\end{aligned}$$

Again, these values can optionally be calculated using the preprocessed values $preproc(e_i)$ instead of e_i for all $i \in [1; n]$. Of course, the adjusted functions described in Section 7.5.3 can be used instead of the standard functions.

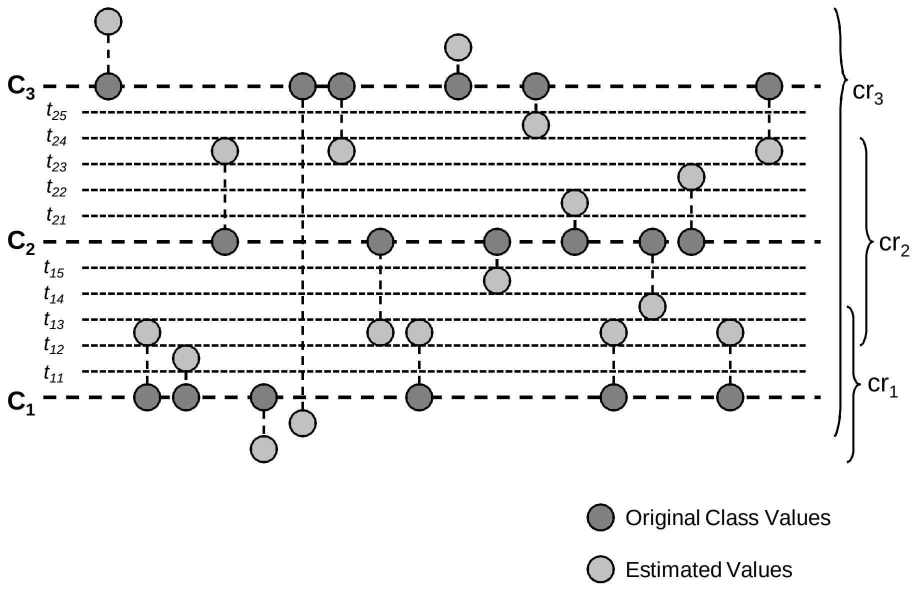


Figure 8.4: Classification example: Original class values, estimated target values and class ranges.

8.2.4.3 Considering Classification Specific Aspects

We propose the consideration of the following classification specific aspects in the evaluation of classifier models:

- The range of the values estimated for each of the given classes,
- how well the classes are separated correctly from each other depending on the choice of appropriate thresholds, and
- the area under ROC curves or, in the case of multi-class classification, the area under sets of MROC curves.

Class Ranges

For calculating the class ranges $cr_1 \ldots cr_c$ we definitively need the sets of estimated values for each class, $ec_1 \ldots ec_c$:

$$\forall (i \in [1;c]) : cr_i = max(ec_i) - min(ec_i) \tag{8.31}$$

and can so calculate the class ranges' contribution cr as

$$cr = \sum_{i=1}^{c} cr_i \cdot w_i \tag{8.32}$$

Figure 8.4 exemplarily displays several samples with original class values C_1, C_2 and C_3; the class ranges result from the estimated values for each class and are indicated as cr_1, cr_2 and cr_3.

Thresholds Analysis

As is indicated in Figure 8.4 we do not only want to consider class ranges but also a more classification-like approach. Between each pair of contiguous classes we set m equally distributed temporary thresholds:

$$\forall(i \in [1; c-1])\forall(k \in [1; m]) : \; t_{i,k} = C_i + k \cdot \frac{C_{i+1} - C_i}{m+1} \tag{8.33}$$

Then, for each threshold we count the numbers of samples which are classified incorrectly; here we also consider a given matrix storing misclassification punishments mcp for each pair of classes giving the misclassification punishment for classifying a sample of class a as class b as $mcp_{a,b}$ for all a and b in $[1; c]$:

$$\forall(i \in [1; c-1])\forall(k \in [1; m])\forall(j \in [1; n]) :$$

$$p(i,k,j) = \begin{cases} mcp_{i,i+1} \cdot \frac{1}{freq_{o_j}} & : \; o_j < t_{i,k} \,\&e_j > t_{i,k} \\ mcp_{i+1,i} \cdot \frac{1}{freq_{o_j}} & : \; o_j > t_{i,k} \,\&e_j < t_{i,k} \\ 0 & : \; else \end{cases} \tag{8.34}$$

$$p(i,k) = \textstyle\sum_{j=1}^{n} p(i,k,j) \tag{8.35}$$

assuming that a sample j is (temporarily) classified as class $(i+1)$ if $e_j > t_{i,k}$ and as class i if $e_j < t_{i,k}$; $freq_a$ is the frequency of class a, i.e. the number of samples that are originally classified as belonging to class a.

The thresholds' contribution to the classifier's fitness, $thresh$, can be now calculated in two different ways: We can consider the minimum sum of punishments for each pair of contiguous classes as

$$thresh = \sum_{i=1}^{c-1} min_{k \in [1;m]} p(i,k) \tag{8.36}$$

or consider all thresholds which are weighted using threshold weights $tw_{1\ldots m}$ as

$$thresh = \sum_{i=1}^{c-1} \frac{1}{\sum_{k=1}^{m} tw_k} \sum_{k=1}^{m} p(i,k) \cdot tw_k \tag{8.37}$$

Normally, we define the threshold weights tw using minimum and maximum weights, weighting the thresholds at near to the original class values minimally and those in the "middle" maximally:

$$tw_1 = tw_{min},\ tw_m = tw_{min};\ tw_{range} = tw_{max} - tw_{min} \tag{8.38}$$

$$m \texttt{ mod } 2 = 0 \Rightarrow \begin{cases} l = m/2 \\ tw_l = tw_{l+1} = tw_{max} \\ \forall(i \in [2; l-1]) : tw_i = tw_{min} + \frac{tw_{range}}{l-1} \cdot (i-1) \\ \forall(i \in [l+1; m-1]) : tw_{m-i+1} = tw_i \end{cases} \tag{8.39}$$

$$m \texttt{ mod } 2 = 1 \Rightarrow \begin{cases} l = (m+1)/2 \\ tw_l = tw_{max} \\ \forall(i \in [2; l-1]) : tw_i = tw_{min} + \frac{tw_{range}}{(m-1)/2} \cdot (i-1) \\ \forall(i \in [l+1; m-1]) : tw_{m-i+1} = tw_i \end{cases} \tag{8.40}$$

(M)ROC Analysis

Finally, we also consider the area under the (M)ROC curves as described in Section 8.2.3.3: For each class we calculate the AUC values for ROC curves and sets of MROC curves (with a given number of thresholds checked for each class), and then we can either use the average AUC or the maximum AUC for each class weighted with the weighting factors already mentioned before:

$$auc = \begin{cases} \sum_{i=1}^{c} AvgAUC(C_i) \cdot w_i & : \texttt{consider average AUCs} \\ \sum_{i=1}^{c} MaxAUC(C_i) \cdot w_i & : \texttt{consider maximum AUCs} \end{cases} \tag{8.41}$$

8.2.4.4 Combined Classifier Evaluation

As we have now compiled all information needed for estimating the quality of a classifier model in GP, $CLASS$, we calculate the final overall quality using respective weighting factors:

$$\begin{array}{ll} a_1 = mee \cdot c_1 & (c_1 = w_{mee}) \\ a_2 = vaf \cdot c_2 & (c_2 = w_{vaf}) \\ a_3 = r^2 \cdot c_3 & (c_3 = w_{r^2}) \\ a_4 = error_{min} \cdot c_4 & (c_4 = w_{err_{min}}) \\ a_5 = error_{max} \cdot c_5 & (c_5 = w_{err_{max}}) \\ a_6 = cr \cdot c_6 & (c_6 = w_{cr}) \\ a_7 = thresh \cdot c_7 & (c_7 = w_{thresh}) \\ a_8 = auc \cdot c_8 & (c_8 = w_{auc}) \end{array} \tag{8.42}$$

$$CLASS(o,e) = \frac{\sum_{i=1}^{8} a_i \cdot c_i}{\sum_{i=1}^{8} c_i}$$

Chapter 9

Incorporation of A Priori Knowledge as Partial Models

9.1 Introduction: A Priori Knowledge and GP

In this section we shall discuss the incorporation of a priori knowledge about the analyzed system into data based system identification using genetic programming.

"A priori" is a term frequently used in statistics as well as in philosophy; it is mostly used to refer to knowledge that is not derived from measurements and experiments. In contrast to this, knowledge that is not known generally but has to be retrieved from experiments is referred to as "a posteriori" knowledge. Both terms' origin is Latin: "A priori" can be translated as "from [what comes] before", "a posteriori" as "from [what comes] later". Thus, everything that is known before experiments is *a priori*, information that has to be derived from experiments is gained *a posteriori*.

A lot of research work has already been done regarding the use of already existing knowledge and known constraints in evolutionary system identification. Keijzer and Babovic ([KB99], [BK00]), for example, describe the design of dimensionally aware GP; here, the fact that physical measurements are generally accompanied by their units of measurement is utilized leading to an extension of GP that considers the information given by the units of measurement. This is done because it is in general not possible for standard GP to guarantee dimensional consistency: Given the units associated to the given data (variables), the model describing the target variable should be a well-formed dimensioned expression; one should not, for example, add meters and seconds [SS01]. In standard GP, of course, also models can be formed that fulfill these dimensional constraints, but obviously the set of dimensionally consistent models is only a very small fraction of the total models space. The introduction of syntactic constraints into GP is for example described in [Gru96], an application of constrained-syntax GP to the search of rules in medical data can be found for example in [BLFM04].

Grammar guided GP, as for example summarized in [SS01], [RS01] and [RS03], can be seen as an approach for coping with syntactic constraints in GP by combining GP with Backus Naur Form (BNF) grammars [Knu64]: BNF grammars use a start symbol $\mathcal{S}$, sets of terminals and non-terminals ($\mathcal{T}$ and $\mathcal{N}$, respectively) and a set of production rules $\mathcal{P}$ that state how non-terminals are to be rewritten into one of their possible derivations until the expression finally only contains terminals[1].

In the following section we shall discuss strategies for the incorporation of specific instead of general knowledge about a system. This means that we summarize strategies for the introduction of (partial) models into the GP based modeling process.

9.2 Introduction of Partial Models into GP Based System Identification

In many modeling problem situations there is at least partial knowledge available about the system's structure. If the whole structure was known, then we would not necessarily need a structural system identification method as GP; but, as already insinuated, we often only know something about a certain part of the system at hand, but not the total system's structure. Examples are shown in Figure 9.1:

(a) In the rather simple case, a sub-system can be modeled using input variables that are all included in the data basis available for the modeling algorithm. I.e., the subsystem's inputs are a subset of the total system's inputs.
The example given in the left part (a) of Figure 9.1 shows the block diagram

[1]Please note that in this case the terms *terminal* and *non-terminal* means something than in the mathematical, system identification oriented way used otherwise in this thesis: In BNF, non-terminals are rewritten until only terminals are left in order to produce correct expressions; in the case of describing mathematical models, non-terminals are functions that expect input values and return values that are calculated using the given inputs, and terminals are terms that do not take any inputs.

of a subsystem that is modeled as a function of input variables X_1, X_3 and X_5; its output, calculated as $(X_1 + X_3)/X_5$, is then used as input for other functions in the description of the total system.

(b) In the more complex case, a part of the system can be modeled using not only system input variables. In this case we have to consider subsystem descriptions that take inputs which are outputs of other parts of the system.
The example given in the right part (b) of Figure 9.1 shows the block diagram of a subsystem that is modeled in the following way: First, the system's inputs X_1 and X_3 and some third system variable are added, and then the result of this addition is divided by some other system variable. This division's result is returned as the subsystem's output and serves as the system's output or as input for other data processing units.

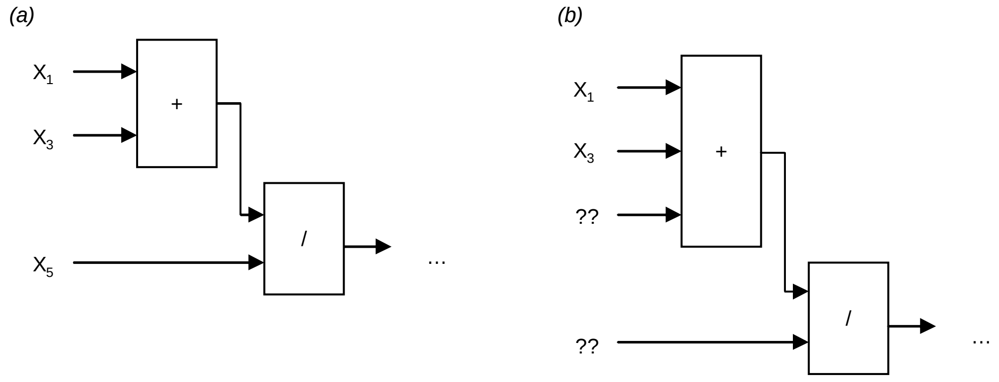


Figure 9.1: A priori knowledge about the structure of a system.

Three possibilities how a priori knowledge can be incorporated into genetic programming based system identification are to be described in the following, namely (1) the introduction of synthetic variables, (2) the intentional seeding of parts of the population, and (3) the introduction of particular terminal and function definition in the functional basis of the GP process.

1. Introduction of synthetic variables:
The most simple way to handle case 1 is to introduce an additional variable into the data base; this new variable's values are calculated according to the subsystem's model. The modeling process is thus able to incorporate this synthetic variable into models for the total system. This procedure is of course applicable and frequently used for any modeling approach.
Still, subsystems as described in modeling case 2 cannot be handled using this approach since not all inputs for the modeled unit are known.

2. Seeding parts of the population:
   A genetic programming specific possibility to handle case 2 is to model the known part of the subsystem as a GP model (formula) m and to inject it into the population intentionally. This injection can be done during the population initialization phase as well as in any other phase of the GP process; in any case a certain number of individuals in the population or of the models created by crossover or mutation has to be replaced by m.
   For the particular example given in the right part (b) of Figure 9.1 this model could be for example `/(+(`X_1`, `X_3`, 0), 1)`; the rest of m's inputs has to be modeled by the evolutionary process, the placeholder terminals `0` and `1` should then be replaced by appropriate subsystem representations. Furthermore, this partial model can be (by crossover) inserted into other models and so become a part of the total system's model. This model is shown in the left part (a) of Figure 9.2.
   Still, this approach comes with two major drawbacks:

   - The models inserted into the population could be assigned very low fitness values and might thus be eliminated out of the population immediately.
   - The models inserted into the population could be assigned very high fitness values, especially when the core of the system is modeled very accurately. The problem here is that these super-individuals could be so dominant in comparison to all other models, and this could have the effect that the population immediately converges to a local optimum so that premature convergence could happen.

3. Introduction of particular definitions in the functional base:
   The most flexible possibility is surely to introduce particular functions and terminals with appropriate parent and child relationship definitions. For modeling a subsystem ss with a set of inputs that are included in the total system's inputs, $\mathbf{i_1}$, and a set of remaining inputs, $\mathbf{i_2}$:

   - Each variable included in $\mathbf{i_1}$ is modeled as a specific terminal definition that refers to the respective variable and enables the GP process to set the respectively allowed sample offsets and coefficients. These specific terminal definitions, in the following referred to as $\mathbf{t_i}$, have to be parameterized with respect to their valid parent definition so that only that functions are allowed as parent functions that are explicitly supposed to use these terminals.
   - ss is then modeled as a function that is programmed according to the knowledge available. The child restrictions are to be set so that only

correct terminals of $\mathbf{t_i}$ are used as inputs at the respective input indices; for all other inputs no specific restrictions have to be defined.

By doing so, any given subsystem can be modeled with optional references to system inputs; by using the respective functions in the genetic programming process, the so modeled a priori knowledge can be incorporated.
This procedure might seem to be a bit cumbersome as it would be easier to program functions that have direct access to the data and to use the variables' values directly without needing additional terminals. Still, we have chosen to stick strictly to the original definition of functions as units processing results of other functions or terminals; this is why this approach has been implemented in this manner in HeuristicLab even though there would not have been a technical reason not to provide functions with access to the data basis.

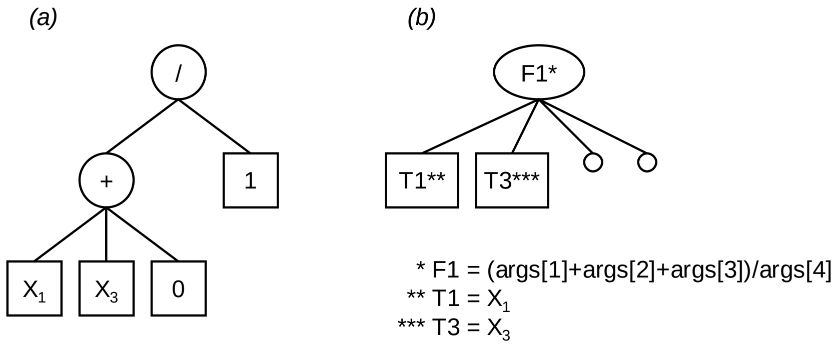


Figure 9.2: Models representing a priori knowledge given in Figure 9.1.

In Section 18 we report on tests incorporating these strategies and their effects on solution quality and population dynamics.

Chapter 10

Local Adaptation Embedded in Genetic Programming

In general, genetic algorithms and genetic programming are considered *global optimization methods*, i.e. their aim is to search the whole search space in an intelligent way in order to find the (or an) optimal solution. In contrast to this, *local optimization methods* are local search algorithms, which means that they move from solution to solution and so search the search space until a solution considered optimal is found (or a time-out condition is fulfilled). Well known examples for local search algorithms are the hill climbing algorithm and tabu search, please see [RN03] and [GL97] for respective explanations and discussions.

In biology, an organism's positive characteristic that has been favored by natural selection is called adaptation [SG99]. This is, in fact, the central concept in evolutionary biology and of course also in evolutionary computation.

In this section we shall summarize local adaptation concepts we have introduced into the genetic programming process, namely parameter optimization as well as model structure pruning.

10.1 Parameter Optimization

Parameter estimation has already been mentioned in connection with classical system identification: After determining and fixing the structure of a model, appropriate parameters have to be estimated on the basis of empirical data.

In GP, the genetic process is supposed to identify the set of relevant variables, the formula structure and appropriate parameters automatically; there are no explicit parameter estimation phases planned in the standard GP process. Furthermore, GP is very flexible regarding function and terminal definitions as well as formula structures; it is not easy to formulate general parameter optimization methods for arbitrary nonlinear model structures.

Still, in GP we have to face the problem that often models with good structures are assigned bad fitness values due to disadvantageous parameters such as coefficients or time lags. This is the reason why we have implemented a parameter optimization method based on evolution strategy (ES) concepts.

Basics of evolution strategies have already been summarized in Section 6.5; here we shall only repeat the main feature of this optimization technique that are relevant in the context of parameter optimization:
In each generation of the execution of an ES, λ individuals (children) are (by mutation and optimal recombination) created out of μ individuals of the current population. Depending on the chosen strategy, the μ members of the new generation's population are selected from all $\mu + \lambda$ candidates (which is referred to as the $(\mu + \lambda)$-ES) or only from the λ children (which is also called the (μ, λ)-ES model). This procedure is repeated until termination criterion is reached, normally a maximum number of iterations or a state in which no more improvement can be reached.

In Section 6.2 we have shown the general form of a polynomial model which is characterized by its order and coefficients:

$$y = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n \tag{10.1}$$

In this case, the optimization of the model's parameters is the task of finding appropriate coefficients $a_0 \ldots a_n$. In the much more general point of view in our GP based approach, the parameters of a model contain a lot more; in fact, all parameter settings of the terminal nodes included in the model are also parameters for the formula which can be optimized without changing the model's structure.

For each terminal the following parameters are to be considered:

- The variable index, i.e. the number of the variable which is referenced.
- The coefficient, a value which can be used for multiplying the referenced variable's value with a given constant; this constant can be either real-valued or integral, and its distribution either uniform (defined by minimum and maximum) or Gaussian (defined by mean and standard deviation).

- The time offset, a value which can be used for referencing to the variable's values shifted by a certain number of samples.

Thus, when it comes to optimizing a model m containing t terminal nodes, we have to consider $3 * t$ parameters that could be manipulated by the optimization method.

As mutation is (besides selection) the most important factor in ES, we shall now discuss how mutation with respect to a model's parameters can be applied. As is explained in further detail in Section 6.5, a parameter σ is used for controlling the strength of mutation; we here see σ simply as the standard deviation of the modification added to the model's parameter values. Thus, each parameter of the model's parameters is modified, where again smaller modifications are more likely than bigger ones; variable index changes are also to be applied rather seldom (for 20% of the terminals, e.g.).

So, the whole parameter optimization procedure we have implemented for optimizing a given model m using the parameters λ and σ is executed in the following way:

1. Collect all terminals of m in $\mathbf{t}$.
2. Create λ copies of m, in the following called mutants.
3. Mutate all λ mutants individually; for each terminal of the mutant models
   - mutate the coefficient,
   - mutate the time offset, and
   - with a rather small probability mutate the variable index.
4. Evaluate all λ mutants.
5. Optionally adjust σ according to Rechenberg's success rule.
6. If any of the mutants is assigned a better quality value than m, then m is replaced by the best mutant, and
   - If the number of iterations has reached a given limit (it_{max}), the algorithm is terminated and m is returned as the optimized version of the originally given formula.
   - Otherwise, the procedure is repeated starting again at step 1.

7. Otherwise, we consider this iteration a failure. If a predefined number of consecutive failures cf_{max} is reached by performing unsuccessfully for cf_{max} times in a row or the number of iterations has reached the given limit it_{max}, then the algorithm is terminated; otherwise the procedure is repeated starting again at step 1.

As we here always work on one particular model which is to be optimized and create λ mutants, this algorithm can be seen as a variant of the $(1 + \lambda) - ES$ algorithm.

Obviously, the main advantage of this algorithm is that it can be applied to any kind of model structure without any restrictions regarding its structure or the given data basis. But, of course the major drawback of this procedure is its immense runtime consumption due to the high number of models that have to be evaluated for improving the parameters of one single model of the GP population. The use of a smaller data set (or the validation set which is normally also smaller than the training data sample) for evaluating the models can help to fight this problem, but still the use of this parameter optimization concept has to be thought out well and the parameters (σ, λ, it_{max} and cf_{max}) set so that the runtime consumption will not get out of hand completely. This parameter optimization method does not have to be applied in every round of the GP process, and also not to all models in the population; partial use can help to control the additional runtime consumption and still use the significant benefits of this procedure.

10.2 Pruning

10.2.1 Basics and Method Parameters

Whenever gardeners and orchardists talk about pruning, then they most probably refer to the act of cutting out dead, diseased or for any other reason unwanted branches of trees or shrubs. Even though this might harm the natural form of plants, pruning is supposed to improve the plants' health in the long run.

In informatics and especially machine learning, this term is used in analogy to describe the act of systematically removing parts of decision trees; regression or classification accuracy is decreased intentionally and thus traded for simplicity and better generalization of the model evolved. Approaches and benefits of the techniques used can be found for example in [Min89], [BB94], [HS95] or [Man97].

Obviously, the concept of removing branches of a tree can be easily transferred to GP, especially when we deal with tree-based genetic programming. Several pruning operators have already been presented for GP, see for example [ZM96], [FP98], [MK00], or more recent publications such as [dN06], [EKK04], [DH02], [FPS06], [GAT06]. In GP, pruning is often considered valuable because it helps to find more general and not over-parameterized programs; it is also referred to as an appropriate anti-bloat technique as described in Section 3.6 or [LP02], e.g.

In the case of fixed functional bases, pruning can also include the detection of really ineffective code or introns, i.e. code fragments that do not contribute to the program's (or, as in our case, model's) evaluation. For example, simply by using basic algebraic analysis, a simplification mechanism for formulas would be able to detect that `-(+(x;4);4)` is equal to `+(+(x;4);-4)` due to basic knowledge about subtraction and addition, and that this is again equal to `+(x;4;-4)`. This then can be easily simplified to `x` as it is easy to implement a simplification program "knowing" that the addition of any value x and its negative counterpart $-x$ is always 0, and that 0 is the neutral element of the addition function.

But, as soon as such a fixed functional basis is not available anymore, things start to become a lot more complicated. We shall here describe pruning methods suitable for use in combination with a flexible and parameterizable set of function and terminal definitions as described in Section 7.3. We hereby try to consider the gain of simplicity as well as the deterioration of the model's quality caused by pruning it:

- The gain of simplicity with respect to the pruning of a model can be calculated by comparing its original tree complexity and the complexity of the pruned structure tree. The complexity of a model m, $c(m)$m, can hereby be equal to the size or the height of the tree structure representing m.
  So, we calculate the model complexity decrease $mcd(m, m_p)$ of a model m and a pruned version of m, m_p, as
  $$mcd(m, m_p) = \frac{c(m)}{c(m_p)} \tag{10.2}$$
  Pruning a model by deleting sub-trees will therefore always result in a mcd value greater than 1 as the original model's complexity (in terms of size or height of the tree structure) will always be greater than the pruned model's complexity.

- The deterioration of a model caused by pruning ($det(m, m_p)$) can be measured by calculating the ratio of the pruned model's quality $q(m_p)$ and the quality of the original formula $q(m)$ as

$$det(m, m_p) = \frac{q(m_p)}{q(m)} \tag{10.3}$$

Thus, if for example the pruned model's fitness value is 10% higher, i.e. worse than the original model's quality with respect to a given evaluation operator, then the resulting deterioration coefficient will be equal to 1.1.

Please note that this approach yields reasonable results only when using a minimization approach, i.e. if better models are assigned smaller quality values as it is for example the case with the MSE function. If the evaluation operator applied behaves reciprocally, i.e. if for example the R^2 or VAF function is used, then the reciprocal value of $det(m, m_p)$, i.e. $\frac{1}{det(m,m_p)}$ is to be used instead.

These measures for the effect of pruning, namely the complexity reduction as well as the quality deterioration, are now used for parameterizing the effective pruning of models:

As we have mentioned already, accuracy is traded for simplicity, and now we are able to quantify this trading aspect. By giving an upper bound for the relation between the coefficients expressing the complexity deterioration and the simplification effects, the pruning mechanism can be limited; we call this composed coefficient $cp(m, m_p)$ and define an upper bound cp_{max} for $cp(m, m_p)$ by demanding that

$$\frac{det(m, m_p)}{mcd(m, m_p)} = cp(m, m_p) \leq cp_{max} \tag{10.4}$$

Thus, we demand that the decrease with respect to the model's quality should not be worse than the simplicity gain multiplied with a certain factor cp_{max}.

Still, there is one major problem with this approach as tremendous loss of quality, as for example an increase of the mean squared error by a factor of 50, might be compensated by replacing a formula m_1 consisting of 60 nodes by one single constant, i.e. a model m_2 with only one node:

$$cp(m_1, m_2) = \frac{det(m_1, m_2)}{mcd(m_1, m_2)} = \frac{\frac{q(m_2)}{q(m_1)}}{\frac{c(m_1)}{c(m_2)}} = \frac{50}{\frac{60}{1}} = \frac{50}{60} < 1 \tag{10.5}$$

So, in order to cope with this potential problem – it is in fact really a problem since we do not want to replace all models with constant terminals – we give a second parameter for the pruning method which limits the quality deterioration, det_{max}, and so demand that

$$det(m, m_p) = \frac{q(m_p)}{q(m)} \leq det_{max} \Leftrightarrow q(m_p) \leq det_{max} * q(m) \qquad (10.6)$$

10.2.2 Pruning a Structure Tree

The actual pruning of a model (with respect to one particular part of the model) in GP is rather easy as it simply consists of removing a sub-tree from the tree structure representing the formula. In the case of pruning the root node the model thereafter is simply a terminal representing the constant 0, otherwise the sub-tree resected is to be replaced by a constant representing the respective parent's neutral element for the respective input index. For example, pruning inputs of an addition results in the replacement of these branches by zeros, whereas children of multiplication functions have to be replaced by constants representing 1.0.

Furthermore, pruning could also include the excision of certain parts of the model, i.e. a part of a tree could be simply cut out and replaced by one of its descendants.

Simple examples are shown in Figure 10.1: In the left part (a) we schematically show the replacement of the second input of an addition resulting in the insertion of the constant 0, in the middle (b) we see the replacement of a multiplication's first input by the constant 1, and in the right part (c) we see possible effects of excising two nodes and replacing them by either of their two descendants.

So, we now know how models are pruned in general as well as what we want a pruning method to achieve. Thus, we here describe two pruning methods we have designed and implemented as operators for HeuristicLab: The first one is an exhaustive implementation that systematically tries to prune the model as much as possible, whereas the second one is inspired by evolution strategies for reducing runtime.

10.2.2.1 Exhaustive Pruning

When applying exhaustive pruning to a given model m we have to proceed in the following way: For each possible subtree up to a given height h_1 we create a copy of m and remove the respective branch. Furthermore, for each internal model fragment (tree) up to a given height h_2 we create a copy of m and cut out the respective

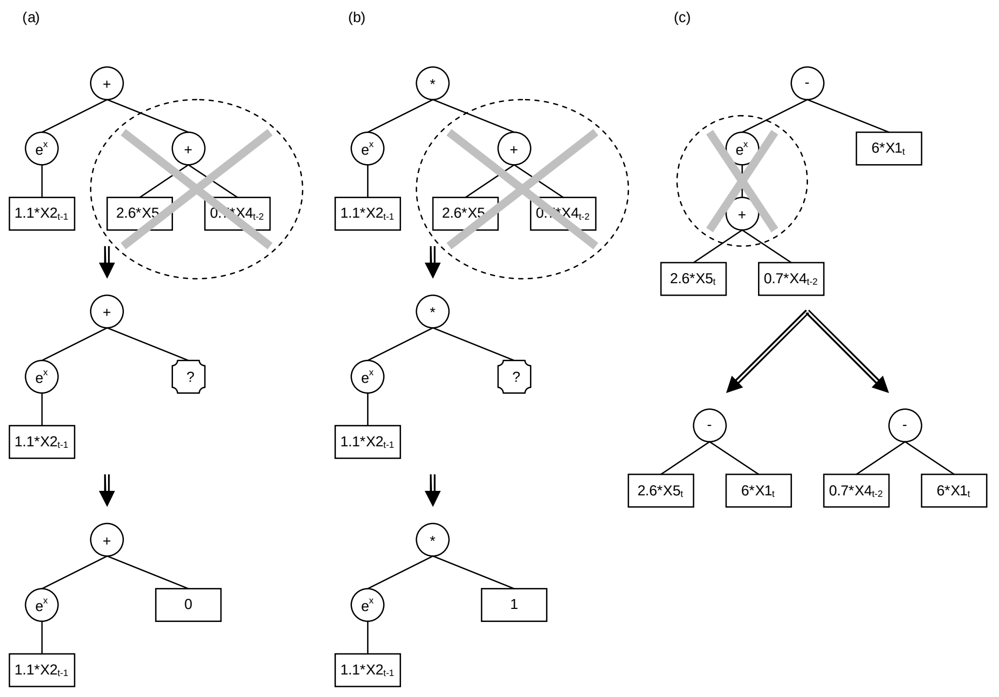


Figure 10.1: Simple examples for pruning in GP.

fragment. After doing so, the resulting pruned models' qualities are calculated and their complexities are checked; if a pruned model meets the requirements regarding maximum deterioration and maximum coefficient of simplification and deterioration, then we go on with the procedure using this pruned formula. This routine is repeated until no more pruned model that meets the given requirements can be produced by deleting branches.

Finally, the algorithm's result is either the minimal model meeting the given requirements, or that model for which the minimal *cp* coefficient is calculated. This decision is controlled by the parameter $minimizeModel$ denoting whether the minimal formula is to be returned or, if this flag is set $false$, the model with the minimal *cp* value is to be considered the result of pruning m.

In a more formal way this exhaustive pruning algorithm is defined in Algorithm 3.

Exhaustive pruning is of course an extremely expensive method with respect to runtime consumption. As an alternative, a general pruning method inspired by evolution strategies is described in the following section.

Algorithm 3 Exhaustive pruning of a model m using the parameters h_1, h_2, $minimizeModel$, cp_{max} and det_{max}.

```
Initialize $m_{curr}$ as clone of $m$,
Evaluate $m$, store calculated fitness in $f$
Calculate complexity of $m$, store result in $c$
Initialize $abort = false$
**while** $not(abort)$ **do**
  Initialize set of pruned models $M$
  Initialize structure tree $t$ as tree representation of $m_{curr}$
  **for each** branch $b$ of $t$ with $height(b) < h_1$ **do**
    Initialize $m_{tmp}$ as clone of $m_{curr}$
    Remove $b'$, the corresponding branch to $b$ in $m_{tmp}$
    Evaluate $m_{tmp}$, store calculated fitness in $f_{tmp}$
    Calculate complexity of $m_{tmp}$, store result in $c_{tmp}$
    Calculate model complexity decrease $mcd = c/c_{tmp}$
    Calculate quality deterioration $det = f/f_{tmp}$
    **if** $det \leq det_{max} \wedge mcd \leq cp_{max}$ **then**
      Insert $m_{tmp}$ to $M$
    **end if**
  **end for**
  **for each** internal sub-tree $st$ of $t$ with $height(st) < h_2$ **do**
    **for each** descendant $d$ of $st$ **do**
      Initialize $m_{tmp}$ as clone of $m_{curr}$
      Replace $st'$, the corresponding part to $st$ in $m_{tmp}$, by $d$
      Evaluate $m_{tmp}$, store calculated fitness in $f_{tmp}$
      Calculate complexity of $m_{tmp}$, store result in $c_{tmp}$
      Calculate model complexity decrease $mcd = c/c_{tmp}$
      Calculate quality deterioration $det = f/f_{tmp}$
      **if** $det \leq det_{max} \wedge mcd \leq cp_{max}$ **then**
        Insert $m_{tmp}$ to $M$
      **end if**
    **end for**
  **end for**
  **if** $M$ is empty **then**
    **return** $m_{curr}$
  **else**
    **if** $minimizeModel$ **then**
      Set $m_{curr}$ to that model in $M$ with minimum complexity value $c$
    **else**
      Set $m_{curr}$ to that model in $M$ with minimum $mcd$ coefficient
    **end if**
  **end if**
**end while**
```

10.2.2.2 ES-Inspired Pruning

As a less runtime consuming pruning method we have designed an ES-inspired pruning method: For pruning a model m, we create λ clones of m and prune those randomly; again, we use parameters h_1 and h_2 that limit the size of the branches and internal subtrees that are excised. All of the so created λ pruned mutants are checked and those, that fulfill the given requirements regarding maximum deterioration and maximum coefficient of simplification and deterioration, are collected. This procedure is then repeated with the best pruned mutant, whereas the best pruned model is again selected as the minimal model or that showing the best coefficient of simplification and deterioration. As soon as this procedure is executed without any success for a given number in a row, the algorithm is terminated.

Algorithm 4 describes this $(1 + \lambda)$-ES-inspired pruning method in a more formal way.

Algorithm 4 ES-inspired pruning of a model m using the parameters λ, $maxUnsuccRounds$, h_1, h_2, $minimizeModel$, cp_{max} and det_{max}.

```
Initialize $m_{curr}$ as clone of $m$,
Evaluate $m$, store calculated fitness in $f$
Calculate complexity of $m$, store result in $c$
Initialize $UnsuccessfulRounds := 0$
Initialize $abort := false$
while $not(abort)$ do
  Initialize set of pruned models $M$
  Initialize structure tree $t$ as tree representation of $m_{curr}$
  for $i = 1 : \lambda$ do
    Set $r$ to random number in $[0; 1[$
    Initialize $m_{tmp}$ as clone of $m_{curr}$
    if $r < 0.5$ then
      Remove $b$, a branch of $m_{tmp}$ with $height(b) < h_1$
    else
      Select $st$, an internal subtree of $t$ with $height(st) < h_2$,
      replace $st$ by a randomly chosen descendant of $d$
    end if
    Evaluate $m_{tmp}$, store calculated fitness in $f_{tmp}$
    Calculate complexity of $m_{tmp}$, store result in $c_{tmp}$
    Calculate model complexity decrease $mcd = c/c_{tmp}$
    Calculate quality deterioration $det = f/f_{tmp}$
    if $det \leq det_{max} \wedge mcd \leq cp_{max}$ then
      Insert $m_{tmp}$ to $M$
    end if
  end for
  if $M$ is empty then
    Increase $UnsuccessfulRounds$
    if $UnsuccessfulRounds = maxUnsuccRounds$ then
      return $m_{curr}$
    end if
  else
    Set $UnsuccessfulRounds := 0$
    if $minimizeModel$ then
      Set $m_{curr}$ to that model in $M$ with minimum complexity value $c$
    else
      Set $m_{curr}$ to that model in $M$ with minimum $mcd$ coefficient
    end if
  end if
end while
```

Chapter 11

Similarity Measures for GP Solutions

Genetic diversity and population dynamics are very interesting aspects when it comes to analyzing GP processes. Measuring the entropy of a population of trees can be done for example by considering the programs' scores (as explained in [Ros95b], e.g.); entropy is there calculated as $-\sum_k p_k \cdot log(p_k)$ (where p_k is the proportion of the population P occupied by population partition k). In [McK00] the traditional fitness sharing concept from the work described in [DG89] is applied to test its feasibility in GP.

In this section we present more sophisticated measures which we have used for estimating the genetic diversity in GP populations as well as among populations of multi-population GP applications. What we use as basic measures for this are the following two functions that calculate the similarity of GP solution candidates or, a bit more specific, in our case formulas represented as structure trees:

- *Evaluation based* similarity estimation compares the sub-trees of two GP formulas with respect to their evaluation on the given training or validation data. The more similar these evaluations are with respect to the squared errors or linear correlation, the higher is the similarity for these two formulas.
- *Structural* similarity estimation directly compares the genetic material of two solution candidates: All possible pairs of ancestor and descendant nodes in formula trees are collected and these collections compared for pairs of formulas. So we can determine how similar the genetic make-up of formulas is without considering their evaluation.

11.1 Evaluation Based Similarity Measures

The main idea of our evaluation based similarity measures is that the building blocks of GP formulas are subtrees that are exchanged by crossover and so form new formulas. So, the evaluation of these branches of all individuals in a GP population can be used for measuring the similarity of two models m_1 and m_2:

For all sub-trees in the structure-tree of model m, collected in t, we collect the evaluation results by applying these sub-formulas to the given data collection *data* as

$$\forall(st_i \in t)\forall(j \in [1;N]) : e_{i,j} = eval(st_i, data) \tag{11.1}$$

where N is the number of samples included in the data collection, no matter if training or validation data are considered.

The evaluation based similarity of models m_1 and m_2, $es(m_1, m_2)$, is calculated by iterating over all subtrees of m_1 (collected in t_1) and, for each branch, picking that subtree of t_2 (containing all sub-trees of m_2) whose evaluation is most “similar” to the evaluation of that respective branch. So, for each branch b_a in t_1 we compare its evaluation e_a with the evaluation e_b of all branches b_b in t_2, and the “similarity” can be calculated using the sum of squared errors (sse) or the linear correlation coefficient:

- When using the sse function, the sample-wise differences of the evaluations of the two given branches are calculated and their sum of squared differences is divided by the total sum of squares tss of the first branch's evaluation. This results in the similarity measure s for the given branches.

$$\overline{e_1} = \frac{1}{N}\sum_{j=1}^{N} e_a[j] \tag{11.2}$$

$$sse = \sum_{j=1}^{N}(e_a[j] - e_b[j])^2; tss = \sum_{j=1}^{N}(e_a[j] - \overline{e_a})^2 \tag{11.3}$$

$$s_{sse}(b_a, b_b) = 1 - \frac{sse}{tse} \tag{11.4}$$

- Alternatively the linear correlation coefficient can be used:

$$\overline{e_a} = \frac{1}{N}\sum_{j=1}^{N} e_a[j]; \overline{e_b} = \frac{1}{N}\sum_{j=1}^{N} e_b[j] \tag{11.5}$$

$$s_{lc}(b_a, b_b) = \left| \frac{\frac{1}{n-1}\sum_{j=1}^{N}(e_a[j] - \overline{e_a})(e_b[j] - \overline{e_b})}{\sqrt{\frac{1}{n-1}\sum_{j=1}^{N}(e_a[j] - \overline{e_a})^2}\sqrt{\frac{1}{n-1}\sum_{j=1}^{N}(e_b[j] - \overline{e_b})^2}} \right| \tag{11.6}$$

No matter which approach is chosen, the calculated similarity measure for the branches b_a and b_b, $sim(b_a, b_b)$, will always be in the interval $[0;1]$; the higher this value becomes, the smaller is the difference between the evaluation results.

As we can now quantify the similarity of evaluations of two given subtrees, we can for each branch b_a in t_a elicit that branch b_x in t_b with the highest similarity to b_a; the similarity values **s** are collected for all branches in t_a and their mean value finally gives us a measure for the evaluation based similarity of the models m_a and m_b, $es(m_a, m_b)$.
Optionally we can force the algorithm to select each branch in t_b not more than once as best match for a branch in t_a for preventing multiple contributions of certain parts of the models.

Finally, this similarity function can be parameterized by giving minimum and maximum bounds for the height and / or the level of the branches investigated. This is important since we can so control which branches are to be compared, be it the rather small ones, rather big ones or all of them.

Algorithm 5 summarizes this evaluation based similarity measure approach.

11.2 Structural Similarity Measures

Structural similarity estimation is, unlike the evaluation based described before, independent of data; it is calculated on the basis of the genetic make-up of the models which are to be compared.

Koza [Koz92] used the term variety to indicate the number of different programs in populations by comparing programs structurally and looking for exact matches. The Levenshtein distance [Lev66] can be used for calculating the distance between trees, but it is considered rather far from ideal ([Kei96], [O'R97], [LP02]); in [EN00] an edit distance specific to genetic programming parse trees was presented which considered the cost of substituting between different node types.

A very comprehensive overview of program tree similarity and diversity measures has been given for instance in [BGK04]. The standard tree structures representation in GP makes it possible to use more fine grained structural measures that consider

Algorithm 5 Calculation of the evaluation based similarity of two models m_1 and m_2 with respect to data base *data*

```
Collect all subtrees of the tree structure of $m_1$ in $B_1$
Collect all subtrees of the tree structure of $m_2$ in $B_2$
Initialize $s := 0$
for each branch $b_j$ in $B_1$ do evaluate $b_j$ on $data$, store results in $e_{1,j}$
for each branch $b_k$ in $B_2$ do evaluate $b_k$ on $data$, store results in $e_{2,k}$
for each branch $b_j$ in $B_1$ do
  Initialize $s_{max} := 0$, $index := -1$
  if $|B_2| > 0$ then
    for each branch $b_k$ in $B_2$ do
      Calculate similarity $s_{tmp}$ as similarity of $b_j$ and $b_k$ using $e_{1,j}$, $e_{2,k}$
      and similarity function $s_{sse}$ or $s_{lc}$
      if $s_{tmp} > s_{max}$ do $s_{max} := s_{tmp}$; $index = k$
    end for
    if $PreventMultipleContribution$ do remove $b_{index}$ from $B_2$
  end if
  $s := s + s_{max}$
end for
return $s/|B_1|$
```

nodes, subtrees, and other graph theoretic properties (rather than just entire trees). In [Kei96], for example, subtree variety is measured as the ratio of unique subtrees over total subtrees and program variety as a ratio of the number of unique individuals over the size of the population; [MH99] investigated diversity at the genetic level by assigning numerical tags to each node in the population.

When analyzing the structure of models we have to be aware of the fact that often structurally different models can be equivalent. Let us for example consider the formulas `*(+(2,X2),+(X3)` and `+(*(X2,X3),*(X3,2))`: As we know about distributivity we know that these formulas can be considered equivalent, but any structure analysis approach taking into account size, shape or parent / child relationships in the structure tree would assign these models a rather low similarity value. This is why we have designed and implemented a method that systematically collects all pairs of ancestor and descendant nodes and information about the properties of these nodes. Additionally, for each pair we also document the distance (with respect to the level in the model tree) and the index of the ancestor's child tree containing the descendant node. The similarity of two models is then, in analogy to the method described in the previous section, calculated by comparing all pairs of ancestors and descendants in one model to all pairs of the other model and averaging the similarity of the respective best matches.

Figure 11.1 shows a simple formula and all pairs of ancestors and descendants included in the structure tree representing it; the input indices as well as the level differences ("level delta") are also given. Please note: The pairs given on the right side of Figure 11.1 are shown intentionally as they symbolize the pairs of nodes with level difference 0, i.e. nodes combined with themselves.

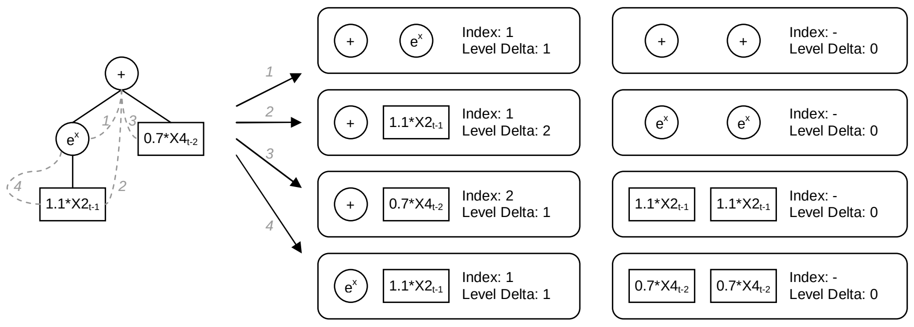


Figure 11.1: Simple formula structure and all included pairs of ancestors and descendants (genetic information items).

We define a *genetic item* as a 6-tuple storing the following information about the ancestor node a and descendant node d:

- $type_a$, the type of the ascendant a
- $type_d$, the type of the descendant d
- δl, the level delta
- *index*, the index of the child branch of a that includes d
- np_a, the node parameters characterizing a
- np_d, the node parameters characterizing d

where the parameters characterizing nodes are represented by tuples containing the following information:

- *var*, the variant (of functions)
- *coeff*, the coefficient (of terminals)

- *to*, the time offset (of terminals)
- *vi*, the variable index (of terminals)

Now we can define the similarity of two *genetic items* gi_1 and gi_2, $sim(gi_1, gi_2)$, as follows:

Most important are the types of the definitions referenced by the nodes; if these are not equal, then the similarity is 0 regardless of all other parameters:

$$\forall(gi_1, gi_2) : gi_1.type_a \neq gi_2.type_a \Rightarrow sim(gi_1, gi_2) = 0 \tag{11.7}$$
$$\forall(gi_1, gi_2) : gi_1.type_d \neq gi_2.type_d \Rightarrow sim(gi_1, gi_2) = 0 \tag{11.8}$$

If the types of the nodes correspond correctly, then the similarity of gi_1 and gi_2 is calculated using the similarity contributors $s_1 \ldots s_{10}$ of the parameters of gi_1 and gi_2 weighted with coefficients $c_1 \ldots c_{10}$.

The differences regarding input index, variant and variable index are not in any way scaled or relativized, their similarity contribution is 1 in the case of equal parameters for both genetic items and 0 otherwise. The differences regarding level difference, coefficient and time offset, on the contrary, are indeed scaled:

- The level difference is divided by the maximum tree height $height_{max}$,
- the difference of coefficients is divided by the range of the referenced terminal definition (in case of uniformly distributed coefficients) or divided by the standard deviation σ (in case coefficients are normally distributed), and
- the difference of the time offsets is divided by the maximum time offset allowed $offset_{max}$.

$$\forall(gi_1, gi_2 : gi_1.type_a = gi_2.type_a \ \&\ gi_1.type_d = gi_2.type_d) :$$

$$s_1 = 1 - \frac{|gi_1.\delta l - gi_2.\delta l|}{height_{max}}, s_2 = \begin{cases} gi_1.index \neq gi_2.index & : \quad 0 \\ gi_1.index = gi_2.index & : \quad 1 \end{cases} \tag{11.9}$$

$$s_3 = \begin{cases} gi_1.np_a.var \neq gi_2.np_a.var & : \quad 0 \\ gi_1.np_a.var = gi_2.np_a.var & : \quad 1 \end{cases} \tag{11.10}$$

$$s_4 = \begin{cases} gi_1.np_d.var \neq gi_2.np_d.var & : \quad 0 \\ gi_1.np_d.var = gi_2.np_d.var & : \quad 1 \end{cases} \tag{11.11}$$

$$\delta c_a = |gi_1.np_a.coeff - gi_2.np_a.coeff| \tag{11.12}$$

$$s_5 = 1 - \begin{cases} isUniform(gi_1.type_a) & : \frac{\delta c_a}{gi_1.type_a.max - gi_1.type_a.min} \\ isGaussian(gi_1.type_a) & : \frac{\delta c_a}{gi_1.type_a.\sigma * 4} \end{cases} \tag{11.13}$$

$$\delta c_d = |gi_1.np_d.coeff - gi_2.np_d.coeff| \tag{11.14}$$

$$s_6 = 1 - \begin{cases} isUniform(gi_1.type_d) & : \frac{\delta c_d}{gi_1.type_d.max - gi_1.type_d.min} \\ isGaussian(gi_1.type_d) & : \frac{\delta c_d}{gi_1.type_d.\sigma * 4} \end{cases} \tag{11.15}$$

$$s_7 = 1 - \frac{|gi_1.np_a.to - gi_2.np_a.to|}{offset_{max}} \tag{11.16}$$

$$s_8 = 1 - \frac{|gi_1.np_d.to - gi_2.np_d.to|}{offset_{max}} \tag{11.17}$$

$$s_9 = \begin{cases} gi_1.np_a.vi \neq gi_2.np_a.vi & : 0 \\ gi_1.np_a.vi = gi_2.np_a.vi & : 1 \end{cases} \tag{11.18}$$

$$s_{10} = \begin{cases} gi_1.np_d.vi \neq gi_2.np_d.vi & : 0 \\ gi_1.np_d.vi = gi_2.np_d.vi & : 1 \end{cases} \tag{11.19}$$

Finally, there are two possibilities how to calculate the structural similarity of gi_1 and gi_2, $sim(gi_1, gi_2)$: On the one hand this can be done in an *additive* way, on the other hand in a *multiplicative* way.

- When using the *additive* calculation, which is the obviously more simple way, $sim(gi_1, gi_2)$ is calculated as the sum of these similarity contributions $s_{1...10}$ weighted using the factors $c_{1...10}$ and, for the sake of normalization of results, divided by the sum of the weighting factors:

$$sim(gi_1, gi_2) = \frac{\sum_{i=1}^{10} s_i \cdot c_i}{\sum_{i=1}^{10} c_i} \tag{11.20}$$

- Otherwise, when using the *multiplicative* calculation method, we first calculate a punishment factor p_i for each s_i (using weighting factors c_i, $0 \leq c_i \leq 1$ for $i \in [1, 10]$) as $p_i = (1 - s_i) \cdot c_i$ and then get the temporary similarity result as

$$sim_{tmp}(gi_1, gi_2) = \prod_{i=1}^{10}(1 - p_i). \tag{11.21}$$

In the worst case scenario we get $s_i = 0$ for all $i \in [1, 10]$ and therefore the worst possible sim_{tmp} is

$$sim_{worst} = \prod_{i=1}^{10}(1 - ((1 - s_i) \cdot c_i)) = \prod_{i=1}^{10}(1 - c_i). \tag{11.22}$$

As sim_{worst} is surely greater than 0 we linearly scale the results to the interval $[0, 1]$:

$$sim(gi_1, gi_2) = \frac{sim_{tmp}(gi_1, gi_2) - sim_{worst}}{1 - sim_{worst}}. \tag{11.23}$$

In fact, we prefer this *multiplicative* similarity calculation method since it allows more specific analysis: By setting a weighting coefficient c_j to a rather high value (i.e., near or even equal to 1.0) the total similarity will become very small for pairs of genetic items that do not correspond with respect to this specific aspect, even if all other aspects would lead to a high similarity result.

Based on this similarity measure it is easy to formulate a similarity function that measures the similarity of two model structures. In analogy to the approach presented in the previous section, for comparing models m_1 and m_2 we collect all pairs of ancestors and descendants (up to a given maximum level difference) in m_1 and m_2 and look for the best matches in the respective opposite model's pool of genetic items, i.e. pairs of ancestor and descendant nodes. As we are able to quantify the similarity of genetic items, we can elicit for each genetic item gi_1 in the structure tree of m_1 exactly that genetic item gi_x in the model structure m_2 with the highest similarity to gi_1; the similarity values $\mathbf{s}$ are collected for all genetic items contained in m_1 and their mean value finally gives us a measure for the structure based similarity of the models m_1 and m_2, $ss(m_1, m_2)$.
Optionally we can force the algorithm to select each genetic item of m_2 not more than once as best match for an item in m_1 for preventing multiple contributions of certain components of the models.

This function is defined in a more formal way using pseudo-code in Algorithm 6.

Obviously, it is possible that some model exactly contains all pairs of genetic items that are also incorporated in another model, but not vice versa. Thus, this similarity measure $ss(m_1, m_2)$ is not symmetric, i.e. $ss(m_1, m_2)$ does not necessarily return the same result as $ss(m_2, m_1)$ for any pair of models m_1 and m_2.

Of course, this similarity concept for GP individuals cannot be the basis of theoretical concepts comparable to those based on GP (hyper)schemata, for example; we do here not want to give any statements about the probability of certain parts of formulas to occur in a given generation. In the presence of mutation or other structure modifying operations (as for example pruning) we are interested in measuring the structural diversity in GP populations; using this structural similarity measure we are able to do so.

Algorithm 6 Calculation of the structural similarity of two models m_1 and m_2

```
Collect all genetic items $m_1$ in $GI_1$
Collect all genetic items $m_2$ in $GI_2$
Initialize $s := 0$
for each branch $gi_j$ in $GI_1$ do
   Initialize $s_{max} := 0$, $index := -1$
   if $|B_2| > 0$ then
      for each genetic item $gi_k$ in $GI_2$ do
         Calculate similarity $s_{tmp}$ as similarity of $gi_j$ and $gi_k$
         if $s_{tmp} > s_{max}$ do $s_{max} := s_{tmp}$; $index = k$
      end for
      if $PreventMultipleContribution$ do remove $gi_{index}$ from $GI_2$
   end if
   $s := s + s_{max}$
end for
return $s/|GI_1|$
```

Chapter 12

Population Dynamics in Genetic Programming

There are several aspects of dynamics in GP populations that can be observed and analyzed. In this section we shall describe those aspects which we have concentrated on and which will also be analyzed for evaluating different algorithmic GP settings on various problem instances:

- In Section 12.1 we describe how we analyze which individuals of the population succeed in passing their genetic information on to the next generation.
- The analysis of variables diversity during the execution of the GP process is described in Section 12.2. Hereby we distinguish between the occurrences of references to variables, i.e. the frequencies of variables in the population, and the impact of variables, i.e. we measure how strongly the evaluation of models is altered when temporarily removing information of variables.
- The diversity of populations with respect to the frequencies of function and terminal definitions is described in Section 12.3.
- In Section 12.4 we finally give a summary of approaches for analyzing the diversity among GP populations using the similarity measures described in Chapter 11. We use these concepts to measure how diverse the individuals of populations are as well as how similar populations of multi-population GP processes become during runtime.

12.1 Parents Analysis

In the context of conventional GAs, parent selection is normally responsible for selecting fitter individuals more often than those that are less fit. Thus, fitter individuals are supposed to pass on their genetic material to more members of the next generation's population.

When using offspring selection, several additional aspects have to be considered. As only those children survive this selection step that perform better than their parents to a certain degree, we cannot guarantee that fitter parents succeed more often than less fit ones.

This is why we document the parent indices of all successful offspring for each generation step. So we can analyze whether all parts of the population are considered for effective propagation of their genetic information, if only better ones or rather bad ones are successful.

Formally, in parent analysis we analyze the genetic propagation of parents **P** to their children **C** calculating the propagation count *pc* for each parent as the number of successful children it was able to produce by being crossed with other parents or mutation:

$$isParent(p,c) = \begin{cases} 1 & : \quad p \in c.Parents \\ 0 & : \quad otherwise \end{cases} \tag{12.1}$$

$$\forall (p \in \mathbf{P}) : pc(p) = \sum_{c \in \mathbf{C}} isParent(p,c) \tag{12.2}$$

In addition, we can optionally weight the propagation count for each potential parent by weighting it with the similarity of the parent and its children (supposing the availability of a similarity function *sim* which can be used for calculating the similarity of solution candidates):

$$\forall (p \in \mathbf{P}) : pc'(p) = \sum_{c \in \mathbf{C}} isParent(p,c) * sim(p,c) \tag{12.3}$$

12.2 Variables Diversity

For measuring the genetic diversity in GP populations with respect to the variables used we have developed the following features that are able to measure the genetic diversity within a population:

- A very simple approach is to calculate an occurrence feature for each data variable (in the case of time series analysis also considering each possible time offset) as the number of models that include the respective variable. Obviously, this approach can easily be transferred to function definitions as well as terminal definitions.
- As a first extension to this approach, the qualities of the models have to be incorporated into this calculation model as for example by multiplying the occurrence values with a weighting factor (which depends on the model's quality in relation to the qualities of all other models which are included in the current generation's population).
- The most informative (and, unfortunately, also most run-time consuming) calculation model takes into account the impact of variables by evaluating all models assuming that all information included in the respective data variables was deleted temporarily. There are several possibilities how to remove information from a variable, for example by replacing all values by the mean value of the respective training data samples or a given constant, by using linear regression for calculating the variables' trend (again using the respective training data samples) or by adding a synthetic Gaussian noise.

In a first step, the relevance *rel* for each variable with respect to each model of the current population has to be calculated; this is done either by frequency analysis or by measuring its impact by evaluating it on modified data bases.

12.2.1 Frequency Based Relevance of Variables

The relevance of a variable (at index i) with respect to a given model m can either be defined as the number of references in this model (calculated using function $freq_1$) to this variable or simply as 1 if there is a reference to this variable and 0 if not ($freq_2$). All terminals t in the model are considered for this.

$$freq_1(i,m) = |\{t : t \in m.terminals \wedge t.VarIndex = i\}| \tag{12.4}$$

$$freq_2(i,m) = \begin{cases} 1: & \exists t: (t \in m.terminals \wedge t.VarIndex = i) \\ 0: & otherwise \end{cases} \tag{12.5}$$

In the case of time series analysis this variable frequency analysis can be extended to the analysis of time lags as variables are possibly referenced using time offsets (as for example in $f(x) = u_{t-2} * v_{t-1}$). Thus, we calculate the frequency based relevance of a variable (at index i) with time offset t with respect to a model m as follows:

$$freq_1(i,t,m) = \begin{array}{l} |\{t : t \in m.terminals \wedge \\ t.VarIndex = i \wedge t.TimeOffset = t\}| \end{array} \tag{12.6}$$

$$freq_2(i,t,m) = \begin{cases} 1: & \exists t: (t \in m.terminals \wedge t.VarIndex = i \\ & \quad \wedge t.TimeOffset = t) \\ 0: & otherwise \end{cases} \tag{12.7}$$

12.2.2 Impact Based Relevance of Variables

For estimating a variable's impact to the evaluation of a model we temporarily replace the values of exactly this variable and evaluate the model on the basis of the resulting manipulated data base. Thus, first replacement strategies have to be designed; we here present methods using averaging, constants, linear regression and additive Gaussian noise:

A given variable (at index i) is replaced without changing any other part of the data basis; we transfer the original data basis *Data* consisting of N variables with n samples each to a manipulated data basis *Data_i* with manipulated variable number i using a given replacement function r as

$$\forall (i \in [1;N]) : Data\_i(r) = [Data_1, \ldots, Data_{i-1}, r(Data_i), Data_{i+1}, \ldots, Data_N] \tag{12.8}$$

The replacement functions introduced here realize the replacement of a given variable using a given constant value (r_{const}), its mean value (r_{mean}), its linear trend (r_{linreg}) and the addition of Gaussian noise (r_{agn}). Whereas the first two methods are rather straightforward (12.9, 12.10), the other two are more complex: For replacing a variable with its linear trend, we calculate the parameters needed for linear regression using the method of minimizing the sum of squared errors [DS98] (12.11 – 12.15), and a random number generator is needed for adding Gaussian noise together with the respective variable's range and a scaling factor σ (12.16, 12.17).

The following equations express this formally for an arbitrary, but fixed variable number i:

$$\forall(j \in [1,n]) : [r_{const}(Data_i, c)]_j = c \tag{12.9}$$

$$\forall(j \in [1,n]) : [r_{mean}(Data_i, c)]_j = \frac{1}{n}\sum_{j=1}^{n} Data_i[j] \tag{12.10}$$

$$\bar{j} = \frac{1}{n}\sum_{j=1}^{n} j = \frac{n+1}{2} \tag{12.11}$$

$$\forall(j \in [1;n]) : y_j = [Data_i]_j ; \bar{y} = \frac{1}{n}\sum_{j=1}^{n}[Data_i]_j = \frac{1}{n}\sum_{j=1}^{n} y_j \tag{12.12}$$

$$b = \frac{\frac{1}{n}\sum_{j=1}^{n}(j-\bar{j})(y_j-\bar{y})}{\frac{1}{n}\sum_{j=1}^{n}(j-\bar{j})^2} = \frac{\sum_{j=1}^{n}(j-\bar{j})(y_j-\bar{y})}{\sum_{j=1}^{n}(j-\bar{j})^2} \tag{12.13}$$

$$a = \bar{y} - b \cdot \bar{j} \tag{12.14}$$

$$\forall(j \in [1;n]) : [r_{linreg}(Data_i)]_j = a + b \cdot j \tag{12.15}$$

$$range_i = max(Data_i) - min(Data_i) \tag{12.16}$$

$$\forall(j \in [1;n]) : [r_{agn}(Data_i, RG, \sigma)]_j = [Data_i]_j + range_i \cdot RG.next() \cdot \sigma \tag{12.17}$$

Now it is possible to calculate the variable's impact with respect to the model m by evaluating the model on the manipulated data set $Data\_i$ and measuring the resulting difference between the original output values and those calculated on the manipulated data. This measurement can be done on the basis of the average absolute difference function $impact_{mad}$, the mean squared difference function $impact_{msd}$, $impact_{cc}$ using the correlation coefficient and $impact_{ff}$ using a (predefined) external fitness function FF.

The first two functions measure the sample-wise difference between the evaluations on the original and manipulated data, respectively:

$$impact_{mad}(i,m) = \frac{1}{n}\sum_{j=1}^{n} |[eval(m, Data\_i)]_j - [eval(m, Data\_i)]_j| \tag{12.18}$$

$$impact_{msd}(i,m) = \frac{1}{n}\sum_{j=1}^{n} ([eval(m, Data\_i)]_j - [eval(m, Data\_i)]_j)^2 \tag{12.19}$$

The correlation coefficient cc is a non-dimensional measure for the linear interrelationship between two variables. Its domain is $[-1;+1]$ with $cc = +1$ indicating a perfect positive and $cc = -1$ a perfect negative linear relation between the signals investigated; if the correlation coefficient equals 0, the variables show no linear correlation at all.

Thus, the correlation based method $impact_{cc}$ for calculating the impact of a variable i with respect to a given model m can be calculated in the following way: The model is on the one hand evaluated on the (whole) manipulated data set yielding X and on the other hand on the original data set yielding Y; within the analysis approach presented here the absolute value of the correlation coefficient of X and Y is returned as the resulting impact value:

$$\forall(j \in [1;n]) : X_j = [eval(m, Data\_i)]_j ; Y_j = [eval(m, Data)]_j \tag{12.20}$$

$$cc(X,Y) = \frac{\frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2} \cdot \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(Y_i - \bar{Y})^2}} \tag{12.21}$$

$$impact_{cc}(i,m) = 1 - |cc(X,Y)| \tag{12.22}$$

Finally, it also seems to be a good idea to calculate the impact of a variable i with respect to a given fitness function FF. The main idea here is that (for example in the context of evaluating classifier models) it is well possible that the manipulation of a certain variable results in significant changes of the output values of models, but the evaluation using some given fitness measure does not change to the same extent (for example because the ratio of correctly classified samples is not changed). So, again the manipulated as well as the original data set are evaluated using the given model m[1]; the resulting output values are compared to the original target values stored in the data base, and the impact factor is calculated using the ratio between the original evaluation and the evaluation on the basis of the manipulated data set:

$$\forall(j \in [1;n]) : X_j = [eval(m, Data\_i)]_j ; Y_j = [eval(m, Data)]_j ; T_j = [Data_i]_j \tag{12.23}$$

$$q = eval(T,X,FF)/eval(T,Y,FF) \tag{12.24}$$

$$impact_{ff}(i,m,FF) = \begin{cases} 1 - \frac{1}{q} & : \quad q \geq 1 \\ 1 - q & : \quad q < 1 \end{cases} \tag{12.25}$$

[1]Of course, here the following issue has to be kept in mind: If the system's target variable is to be analyzed and therefore manipulated, the evaluation of the model using the given fitness function still has to be done on the original (not manipulated) values of this variable.

12.2.3 Weighting of Variables Relevance Estimations

Before calculating the overall relevance of a variable with respect to a population of models M it is possible to calculate weighting factors for each model depending on its quality:

$$\forall(j \in [1, |M|]) : q_j = quality(m_j); m_j \in M \tag{12.26}$$

$$q_{min} = min_{j\in[1,|M|]}(q_j); q_{max} = max_{j\in[1,|M|]}(q_j) \tag{12.27}$$

$$\forall(j \in [1, |M|]) : weighting(m_j) = 1 - \frac{q_j - q_{min}}{q_{max} - q_{min}} \tag{12.28}$$

12.2.4 Calculating the Relevance of Variables in Populations

Finally, for each variable the (potentially weighted) relevance values calculated for each model are summarized and so give a measure for the overall relevance of this specific signal. By calculating this relevance measure for all variables we finally get an analysis for the genetic diversity within the given population. So, on the basis of all relevance values for each variable (with index i) on each model m (12.29) and after possibly weighting them (12.30) we calculate the relevance of a variable with index i with respect to a whole population P as given in (12.31):

$$rel(i, m) = \left\{ \begin{array}{rcl} freq(i, m) & : & frequencyBasedAnalysis \\ impact(i, m) & : & otherwise \end{array} \right. \tag{12.29}$$

$$rel'(i, m) = \left\{ \begin{array}{rcl} rel(i, m) \cdot weighting(m) & : & weightingActivated \\ rel(i, m) & : & otherwise \end{array} \right. \tag{12.30}$$

$$rel(i, P) = \sum_{m \in P} rel'(i, m) \tag{12.31}$$

12.3 Functions and Terminals Diversity

Another possibility to measure the diversity in GP populations in the context of data based structure identification is to monitor the frequency or the impact of function and terminal definition on the models of the population. For doing so we have designed the following features:

- A very simple approach is to calculate occurrences of each function or terminal definition as the number of models that include respective references.
- As a first extension to this approach, the qualities of the models have to be incorporated into this function or example by multiplying the occurrence values with a weighting factor (which again depends on the models' quality in relation to the quality of all other models which are included in the current generation's population).
- The most informative (and, unfortunately, also most run-time consuming) calculation model takes into account the impact of functions and terminals by evaluating all models after systematically replacing function and terminal definitions by dummy values or randomly chosen other definitions of the same kind.

12.3.1 Frequency Based Relevance of Functions and Terminals

The relevance of a function or terminal definition d with respect to a given model m can either be defined as the number of references in this model (calculated using function $freq_1$) to this definition or simply as 1 if there is a reference to it and 0 if not ($freq_2$). All nodes n in the model are considered for this.

$$freq_1(d,m) = |\{n : n \in n.nodes \wedge n.def = d\}| \tag{12.32}$$

$$freq_2(d,m) = \begin{cases} 1 : & \exists n : (n \in m.nodes \wedge n.def = d) \\ 0 : & otherwise \end{cases} \tag{12.33}$$

12.3.2 Impact Based Relevance of Functions and Terminals

Estimating the impact of a certain function or terminal definition d to a model m is not as straight-forward as for example estimating the relevance of a variable for a model. The main idea of the approach presented here is that the definition is temporarily removed from the model and the model is re-tested; the more this changes the evaluation of the model, the higher is the impact of the temporarily removed definition.

In principle, there are two possibilities how to remove a definition d from a model:

- First, each node referencing d could be neutralized, i.e. replaced by a constant node representing the node's parent's neutral element (for the respective input index). In case of neutralizing the root node, the constant 0 is chosen for replacing the node. We call the modified version of m, in which all nodes referencing d are replaced by neutral constants, $r_n(m, d)$:

$$m' = r_n(m, d) :$$

$$\forall(n \in m') : (n.def = d) \Rightarrow n.def := const,$$

$$n.coeff = \begin{cases} n.parent.neutral : & n.paren \neq null \\ 0 : & otherwise \end{cases} \tag{12.34}$$

- Secondly, all references to d could be replaced by randomly chosen other definitions of the same kind, i.e. a terminal definition is replaced by a randomly chosen other terminal and a function by a randomly chosen other function definition. We call the modified version of m, in which all nodes referencing d are modified so that they reference randomly chosen other functions or terminals, $r_{rcod}(m, d)$:

$$m' = r_n(m, d) :$$

$$isTerminal(d) \Rightarrow \forall(n \in m') :$$

$$(n.def = d) \Rightarrow n.def := term, isTerminal(term), d \neq term \tag{12.35}$$

$$isFunction(d) \Rightarrow \forall(n \in m') :$$

$$(n.def = d) \Rightarrow n.def := func, isFunction(func), d \neq func \tag{12.36}$$

  Please note that this modification can have the effect that a node might reference too few or too many child nodes because the new function definition's arity parameters might differ from those of the function used formerly. In this case children of the respective node have to be removed from or added to the respective node so that structural integrity is assured before evaluating the modified model.

After modifying m we can compare the values calculated by the original model to those calculated using m'; similar to the approaches described in Section 12.2.2 this measurement can be done on the basis of the average absolute difference function $impact_{mad}$, the mean squared difference function $impact_{msd}$, $impact_{cc}$ using the correlation coefficient and $impact_{ff}$ using a (predefined) external fitness function FF.

Let us now assume that the modified model's evaluation on the data basis *data* (storing n samples) yields values stored in X, and that the original model's evaluation yields values stored in Y:

$$\forall(j \in [1;n]) : X_j = [eval(m', data)]_j; Y_j = [eval(m, data)]_j \tag{12.37}$$

The first two functions measure the sample-wise difference between the evaluations on the original and manipulated data, respectively:

$$impact_{mad}(d, m) = \frac{1}{n}\sum_{j=1}^{n} |[Y_j - X_j| \tag{12.38}$$

$$impact_{msd}(d, m) = \frac{1}{n}\sum_{j=1}^{n} (Y_j - X_j)^2 \tag{12.39}$$

Using Equation 12.21 we can calculate the correlation coefficient based impact of definition d as

$$impact_{cc}(d, m) = 1 - |cc(X, Y)| \tag{12.40}$$

Finally, it also seems to be a good idea to calculate the impact of a definition d with respect to a given fitness function FF. The main idea here is again that it is well possible that the manipulation of a certain variable results in significant changes of the output values of models, but the evaluation using some given fitness measure does not change to the same extent. So, again the manipulated as well as the original models are evaluated using the given data; the resulting output values are compared to the original target values stored in the data base, and the impact factor is calculated using the ratio between the original evaluation and the evaluation on the basis of the manipulated data set:

$$\forall(j \in [1;n]) : X_j = [eval(m', data)]_j; Y_j = [eval(m, data)]_j; T_j = [data_{target}]_j \tag{12.41}$$

$$q = eval(T, X, FF)/eval(T, Y, FF) \tag{12.42}$$

$$impact_{ff}(d, m, FF) = \begin{cases} 1 - \frac{1}{q} & : \quad q \geq 1 \\ 1 - q & : \quad g < 1 \end{cases} \tag{12.43}$$

where $data_{target}$ is the target variable.

12.3.3 Calculating the Relevance of Functions and Terminals

Finally, for each function or terminal definition the (potentially weighted) relevance values calculated for each model are summarized and so give a measure for the overall relevance of this specific definition. By calculating this relevance measure for all definitions we finally get an analysis for the genetic diversity within the given population; we then use the relevance values for each definition d_i on each model m (12.44) and optionally weight them (12.45) using the weighting equations (12.26) and (12.28). Finally, we calculate the relevance of a definition d_i with respect to a whole population P as given in (12.46).

$$rel(d,m) = \begin{cases} freq(d,m) & : \; frequencyBasedAnalysis \\ impact(d,m) & : \; otherwise \end{cases} \tag{12.44}$$

$$rel'(d,m) = \begin{cases} rel(d,m) \cdot weighting(m) & : \; weightingActivated \\ rel(d,m) & : \; otherwise \end{cases} \tag{12.45}$$

$$rel(d,P) = \sum_{m \in P} rel'(d,m) \tag{12.46}$$

12.4 Genetic Diversity

In this section we finally describe the measures which we use to monitor the diversity and population dynamics with respect to the genetic make-up of solution candidates. We hereby use the similarity measures described in Chapter 11; either the evaluation based similarity function es or the structural similarity function ss can be used:

$$sim(m_1,m_2) : \begin{cases} sim(m_1,m_2) := es(m_1,m_2) \quad [11.1] \\ \vee \\ sim(m_1,m_2) := ss(m_1,m_2) \quad [11.2] \end{cases} \tag{12.47}$$

As both of these similarity functions are not symmetric, we can alternatively use the mean value of the two possible similarity calls and so define a symmetric similarity measurement:

$$symmetricAnalysis \Rightarrow sim(m_1,m_2) = \frac{sim(m_1,m_2) + sim(m_2,m_1)}{2} \tag{12.48}$$

12.4.1 In Single-Population GP

In the context of single-population GP we are mainly interested in the similarity among the individuals of the population: For each model m of the population P

we calculate the mean and the maximum similarity with all other individuals in the population:

$$meanSim(m, P) = \frac{1}{|P| - 1} \sum_{m2 \in P, m2 \neq m} sim(m, m2) \tag{12.49}$$

$$maxSim(m, P) = max_{(m2 \in P, m2 \neq m)}(sim(m, m2)) \tag{12.50}$$

The mean values of all individuals' similarity values are used for calculating the mean and maximum similarity measures for populations:

$$meanSim(P) = \frac{1}{|P|} \sum_{m \in P} meanSim(m, P) \tag{12.51}$$

$$maxSim(P) = \frac{1}{|P|} \sum_{m \in P} maxSim(m, P) \tag{12.52}$$

12.4.2 In Multi-Population GP

The concepts of parallel evolution of populations in genetic algorithms, which have been summarized in Chapter 5, can of course also be used in data based modeling with genetic programming. In this context we can apply the population diversity analysis for each population separately; in the following we describe a multi-population specific diversity analysis.

Basically, a model m is compared to all models in another population P' which does not include m, and $multiPopSim(m, P')$ is equal to the maximum of the so calculated similarity values:

$$m \notin P' \Rightarrow multiPopSim(m, P') = max_{(m2 \in P')}(sim(m, m2)) \tag{12.53}$$

So we can calculate the multi-population similarity of a model with respect to a set of populations PP as the average *multiPopSim* of the model to all populations except for the "own" one:

$$m \in P \& P \in PP \Rightarrow PP' = P' : P' \in PP \& P' \neq P, \tag{12.54}$$

$$multiPopSim(m, PP) = \frac{1}{|PP'|} \sum_{P' \in PP'} multiPopSim(m, P'), \tag{12.55}$$

Finally, a population's *multiPopSim* value is equal to all its models' multi-population similarity values with respect to the whole set of populations:

$$multiPopSim(P, PP) = \frac{1}{|P|} \sum_{m \in P} multiPopSim(m, PP) \tag{12.56}$$

Chapter 13

GP in Volatile Environments: On-Line and Sliding Window GP

13.1 On-Line GP Based System Identification

Thanks to the fact that the GP process is executed periodically, the insertion of an additional stage can be designed and implemented quite easily. As is graphically shown in Figure 13.1, we have added an additional phase to the standard GP cycle: Before the next generation of solution candidates is produced, possibly available new data are collected from a predefined data source (e.g., a file as in the case of our prototypical implementation).

One of the major advantages of this approach is that the benefits of evolutionary computation (namely the combination of directed and undirected search strategies as well as the use of a certain amount of randomness) are combined with concepts of on-line knowledge discovery and data mining. As described in further detail in this section, this modeling method can be used as an alternative to existing on-line modeling and identification methods that are for example used in industrial fault detection and identification programs.

With respect to the measured data, the algorithm is able to adapt its behavior as new identification data are available: Since all individuals of a GP algorithm's population have to be evaluated every generation, the corresponding data set can be modified after every generation step. This of course means a change of the algorithm's environment and is likely to influence the GP process in several (maybe unforeseen) ways. But since structural identification anyway assumes an underlying

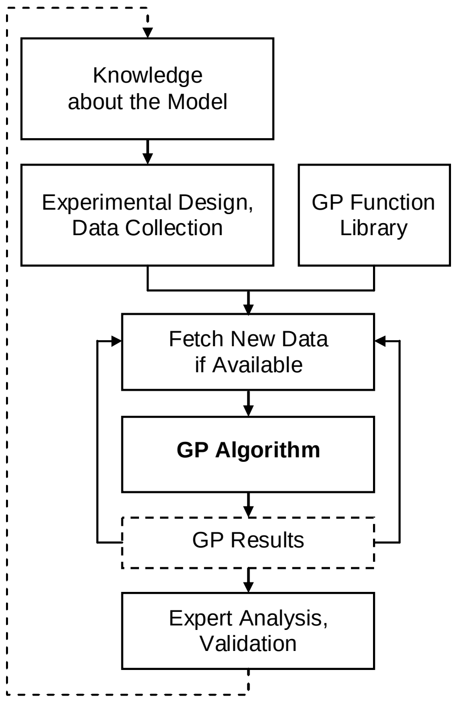

Knowledge about the Model
Experimental Design, Data Collection
GP Function Library
Fetch New Data if Available
GP Algorithm
GP Results
Expert Analysis, Validation

Figure 13.1: Workflow of the on-line GP process.

concept of the investigated system, this changing of environment is expected to have rather positive than negative effects.

Last, but surely not least we strongly take advantage of the fact that instead of using standard implementations of the genetic algorithm as underlying GP algorithm, GP using offspring selection (as described in Section 4.2) is applied. This hybrid GA/GP variant (in our older publications often referred to as the SASEGASA whose most important features have been explained in Section 5.1.6) uses an enhanced selection model which is designed to directly control genetic drift within the population by advantageous self-adaptive selection pressure steering. Additionally, this new selection model makes it possible to detect and hopefully avoid premature convergence which is generally quite a critical issue in GAs.

As already explained in Section 4.2, a very essential question about the general performance of GAs and especially GP is, whether or not good parents are able to produce children of comparable or even better fitness. In natural evolution, this is almost always true. For GAs, this property is not so easy to guarantee and for GP it is a matter of principle that many crossover and mutation results cause counterproductive solution candidates. In order to overcome this drawback, the basic idea of OS is to consider not only the fitness of the parents in order to produce a child for the ongoing evolutionary process: Additionally, the fitness value of the evenly produced child is compared to the fitness values of its own parents; basically the child is accepted as a candidate for the further evolutionary process if and only if the reproduction operator was able to produce a child that could outperform the fitness of its own parents. This strategy guarantees that evolution is continued mainly with crossover results that were able to mix the properties of their parents in an advantageous way which is a very essential aspect for the preservation of essential genetic information stored in many individuals (which might not be the fittest in the sense of individual fitness).

Evolutionary techniques (especially GAs and GP) have often been and are still considered not suitable for on-line identification: "For an off-line process, a genetic programming method could be utilized to 'evolve' the function that best represents the system dynamics. This is an attractive approach because the actual structure of the dynamic equations would be revealed (and the parameters optimized in the process). Unfortunately, evolutionary programming techniques are ill-suited for on-line learning." [Ell98] As we demonstrate in Chapter 16, this widespread opinion has to be reconsidered since the proposed GP-based method is indeed able to evolve suitable models (at least for mechatronical systems) on-line.

Nevertheless, the authors are aware of the fact that the proposed method cannot operate "real-time" in the sense of responding to stimuli within some small upper limit of response time (as, e.g., milli- or microseconds). Due to the fact that the acquisition of new measured data can only be performed after completing a whole generation step of the GP process[1], any GP based method can respond to inputs surely not within milli- or microseconds, but at least within seconds (depending on the time needed to compute a whole generation step).

This approach has been originally presented in [WEA+05], [WAW05a] and [WEA+06].

[1]This is simply because otherwise individuals that have to be compared to each other always have to be evaluated on the basis of the same environmental conditions, i.e. the same target and input signal values.

13.2 Sliding Window Behavior in GP

13.2.1 Basics

The idea of sliding window behavior in computer science is not novel; in machine learning, drifting concepts are often handled by moving the scope (of either fixed or adaptive size) over the training data (see for example [WK96] or [HSD01]). The main idea is the following: Instead of considering all training data for training models (in the case of GP, for evaluating the models produced), the algorithm initially only considers a part of the data. Then, after executing learning routines on the basis of this part of the data, the range of samples under consideration is shifted by a certain offset. Thus, the window of samples considered is moved, it slides along the training data; this is why we are talking about sliding window behavior.

When it comes to GP based structure identification, sliding window approaches are not all too common; in general, the method is seen as a global optimization method working on a set of training samples (which are completely considered by the algorithm within the evaluation of solution candidates). On the contrary, GP is often even considered as an explicitly off-line, global optimization technique. Nevertheless, during research activities in the field of on-line system identification described in the previous section, we discovered several surprising aspects. In general, on-line GP was able to identify models describing a diesel engine's NO_x emissions remarkably fast ([WEA$^+$05], [WAW05a]); the even more astonishing fact was that these models were even less prone to overfitting than those created using standard methods. After further test series and reconsidering the basic algorithmic processes, these facts did not seem to be surprising to us anymore: On the one hand, especially the fact that the environment, i.e. the training data currently considered by the algorithm, is not constant but rather changing during the execution of the training process, contributes positively to the models' quality, it obviously decreases the threat of overfitting. On the other hand, the interplay of a changing data basis and models created using different data also seems to be contributing in a positive way. As the on-line algorithm evaluates models using (new) current training data forgetting samples that were recorded in the beginning, those "old" data are really forgotten from the algorithm's point of view, but the models created on the basis of these old data are still present. The behavior that results out of this procedure is more or less that several possible models that explain the first part of the data are created, and as the scope is moved during the algorithm's execution, only those models are successful that are also able to explain "new" training data.

So, the most self-evident conclusion was that these benefits of online training should be transferred to off-line training using GP. Obviously, this directly leads us to sliding window techniques.

13.2.2 Selection Pressure as Window Moving Trigger

One of the most important problem independent concepts used in our implementation of GP-based structure identification is offspring selection, an enhanced selection model that has enabled genetic algorithms and genetic programming implementations to produce superior results for various kinds of optimization problems.

As in the case of conventional GAs or GP, offspring are generated by parent selection, crossover, and mutation. In a second (offspring) selection step (as it is used in our GP implementation), only those children become members of the next generation's population that outperform their own parents, all other ones are discarded. The algorithm therefore repeats the process of creating new children until the number of successful offspring is sufficient to create the next generation's population. Within this selection model, selection pressure is defined as the ratio of generated candidates to the population size:

$$SelectionPressure = \frac{\mid Solution\ \ candidates\ \ created\ \mid}{\mid Successful\ \ solution\ \ candidates\ \mid}$$

The higher this value becomes, the more models have to be created and evaluated in order to produce a sufficient number of models that are supposed to form the next generation's population. In other words, this selection pressure is a value giving a measure of how hard it is for the algorithm to produce a sufficient number of successful solution candidates.

The proposed idea is to initially reduce the amount of data that is available for the algorithm as identification data. As the identification process is executed, better and better models are created which leads to a rise of the selection pressure; as soon as the selection pressure reaches a predefined maximum value, the limits of the identification data are shifted and the algorithm goes on considering another part of the available identification data set. This procedure is then repeated until the actual training data scope has reached the end of the training data set available for the identification algorithm, i.e. when all data have been considered. By doing so, the algorithm is (following the considerations formulated in the previous section) even less exposed to overfitting, and due to the fact that the models created are evaluated on much smaller data sets we also expect a significant decrease of runtime consumption.

In Algorithm 7 we give a sketch of the sliding window GP based structure identification process incorporating offspring selection. The standard GP parameters (as, for example, population size, mutation rate and crossover operator combinations) are hereby omitted; we only describe the sliding window specific process modifications. The $WindowStartSize$ parameter gives the initial size of the current data window; as soon as the current selection pressure reaches $MaximumSelectionPressure1$, the window is moved by $WindowStepSize$ samples; the $MaximumWindowSize$ parameter specifies the maximum size of the current training data scope. This procedure is repeated until the end of the data set is reached; in the end, the process stops as soon as the second maximum selection pressure parameter value is reached (which does not necessarily have to be the same as the first maximum selection pressure value).

Algorithm 7 The sliding window based GP structure identification process.

```
function Model = SlidingWindowGPStructId (TrainingData, FunctionalBasis,
      WindowStartSize, WindowStepSize, MaximumWindowSize,
      MaximumSelectionPressure1, MaximumSelectionPressure2)
  $index1 = 1$, $index2 = WindowStartSize - 1$
  $InitializeModelsPool$
  while $index2 <= Data.Length$ do
    while $CurrentSelectionPressure <= MaximumSelectionPressure1$ do
      Perform GP-based structure identification using the given $TrainingData$
      in the interval $[index1; index2]$
    end while
    $index2 = Min((index2 + WindowStepWidth), Data.Length)$
    $index1 = Max((index2 - MaximumWindowSize + 1), 0)$
  end while
  $index2 = Data.Length$
  $index1 = Max((Data.Length - MaximumWindowSize + 1), 0)$
  while $CurrentSelectionPressure <= MaximumSelectionPressure2$ do
    Perform GP-based structure identification using the given $TrainingData$
    in the interval $[index1; index2]$
  end while
return Current best model
```

This approach has been originally presented in [WAW07b]; test results are to be summarized and analyzed in Chapter 16.

Part II

Empirical Studies

Chapter 14

Time Series Analysis

14.1 Designing Virtual Sensors for Emissions of a Diesel Engine

The first research work of the Heuristic and Evolutionary Algorithms Laboratory in the area of system identification using GP was done in cooperation with the Institute for Design and Control of Mechatronical Systems (DesCon) at JKU Linz, Austria. The framework and the main infrastructure was given by DesCon who maintain a dynamical motor test bench (manufactured by AVL, Graz, Austria) shown in Figure 14.1. A BMW diesel motor is installed on this test bench, and a lot of parameters of the ECU (engine control unit) as well as engine parameters and emissions are measured; for example, air mass flows, temperatures and boost pressure values are measured, nitric oxides (NO_x, to be described later) are measured using a Horiba Mexa 7000 combustion analyzer, and an opacimeter is used for estimating the opacity of the engine's emissions (in order to measure the emission of particulate matters, i.e. soot).

During several years of research on the identification of NO_x and soot emissions, members of DesCon have tried several modeling approaches, some of them being purely data based as for example those using artificial neural networks (ANNs). Due to rather unsatisfactory results obtained using ANNs, the ability of GP to produce reasonable models was investigated in pilot studies; we a are here once again thankful to Prof. del Re for initiating these studies.

In this context, our goal is to use system identification approaches in order to create models that are designed to replace or support physical sensors; we want to

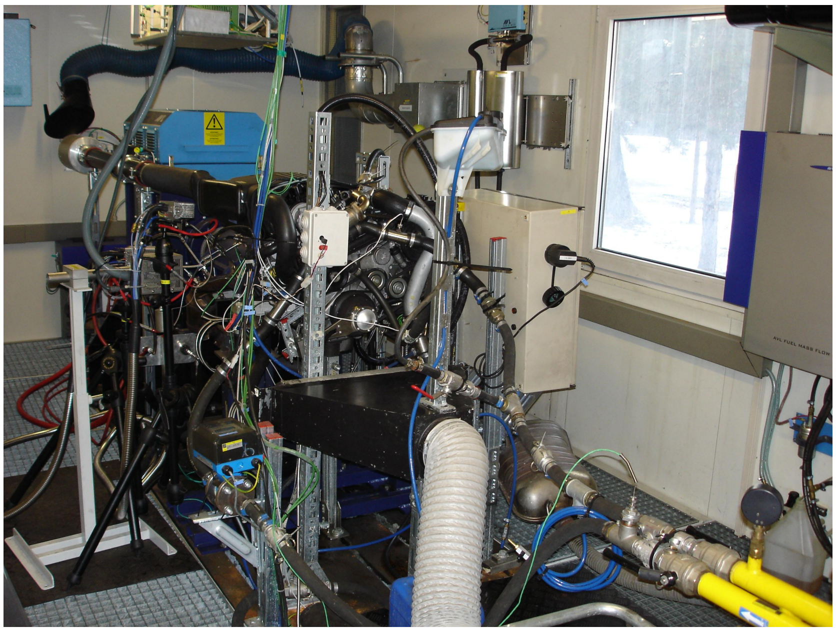


Figure 14.1: Dynamic diesel engine test bench at the Institute for Design and Control of Mechatronical Systems, JKU Linz.

have models that can be potentially used instead of these physical sensors (which can be damageable or simply expensive). This is why we are here dealing with the design of so-called *virtual sensors*.

14.1.1 Designing Virtual Sensors for Nitric Oxides (NO_x)

In general, being able to predict NO_x emissions on-line (i.e., during engine operation) would be very helpful for low emissions engine control. While NO_x formation is widely understood (see for example [dRLF$^+$05] and the references given therein), the computation of NO_x turns out to be too complex and - at the moment - not easy to be used for control. The reason for this is that in theory it would be possible to calculate the engine's NO_x emissions if all relevant parameters (pressures, temperatures, ...) of the combustion chambers were known, but (at least at the moment) we are not able to measure all these values.

As already mentioned above, ANNs have been used for data based modeling of NO_x emissions of a BMW diesel engine. These results were not very satisfying, as is for example documented in [dRLF$^+$05]: Even though modeling quality on training data was very good, the model's ability to predict correct values for operating points not included in the training data was very poor.

We therefore designed and implemented a first GP approach based on HeuristicLab 1.0, preliminary results were published in [WAW04a] and [WAW04b]. In [WAW04b] we documented the ability of GP using offspring selection to produce reasonable models for NO_x, including lots of statistics showing that the results obtained applying rigid offspring selection were significantly better than those obtained without using OS or even OS with less strict parameter settings, i.e. lower success ratio and comparison factor parameters.

NO_x values were recorded by DesCon members following the standard procedure defined by the Federal Test Procedure (FTP); a whole standardized test run is therefore called a FTP cycle. FTP tests were executed on the DesCon test bench in two different ways as it is possible to activate or to deactivate exhaust gas recirculation (EGR). In principle, recirculating a portion of an engine's exhaust gas back to the engine cylinders is called EGR; the incoming air is intermixing with recirculated exhaust gas, which lowers the adiabatic flame temperature and reduces the amount of excess oxygen (at least in diesel engines). Furthermore, the peak combustion temperature is decreased; since the formation of NO_x progresses much faster at high temperatures, EGR can also be used for decreasing the generation of NO_x. Further information about EGR and its effects on the formation of NO_x can for example be found in [Hey88] and [vBS04].

We shall therefore here take a closer look at the following two modeling tasks:

- Situation (1): Use data recorded with deactivated EGR;
- Situation (2): Use data recorded with activated EGR.

In both cases the data were recorded at 20 Hz, the execution of the cycles took approximately 23 minutes. In total, 33 variables are recorded; we do here not give a total overview of the statistic parameters of these variables but rather restrict ourselves to the linear correlation of the input variables to the target variable: All linear correlations[1] of the potential input variables and the target variable NO_x are summarized in Table 14.1; all variables were filtered using a median filter of order 5[2] before calculating the correlation coefficients.

[1]We here use the same standard formula for calculating linear correlation coefficients of time series as described in Section 11.1.

[2]Applying a median filter means that a moving window is shifted over the data and all samples are replaced by the median value of their respective data environment. For calculating the filtered value y_i using median filtering of order 5 we collect the original values x_{i-2}, x_{i-1}, x_i, x_{i+1} and

Table 14.1: Linear correlation of input variables and the target values (NO_x).

Variable	*Correlation Coefficient*		*Variable*	*Correlation Coefficient*	
	Situation (1)	*Situation (2)*		*Situation (1)*	*Situation (2)*
$time$	-0.141	-0.129	$alpha$	0.437	0.462
CO_2	0.477	0.941	COH	0.099	0.259
COL	0.222	0.390	KW_VAL	0.763	0.853
M_T01F	0.414	0.515	ME_MES1	0.408	0.416
ME_MES2	0.460	0.488	ME_MES3	0.043	0.054
ME_MES4	0.000	0.000	ME_MES5	-0.024	-0.007
ME_MES6	0.000	0.000	ME_MES7	-0.092	0.015
ME_MES8	0.492	0.451	ME_MES9	0.660	0.592
ME_MES10	0.135	-0.133	ME_MES11	0.449	0.532
ME_MES12	0.253	0.321	ME_MES13	-0.052	0.376
ME_MES14	0.091	0.101	ME_MES15	0.364	0.314
ME_MES16	0.392	0.478	ME_MES17	-0.438	-0.470
N_MOTOR	0.404	0.413	OPA_OPAC	0.248	0.419
T_EXH	0.347	0.474	T_LLNK	0.133	0.004
T_LLVK	0.531	0.315	T_OIL	0.096	-0.181
THC_VK	-0.074	0.205	TWA	0.149	0.064

Obviously, activating EGR significantly increases the correlation of NO_x and all exhaust variables such as CO_2 or THC, for example.
So, in addition to this, the next question is whether to incorporate gas emissions as for example $\mathrm{CO_2}$ in the modeling process; of course, estimating NO_x is a lot easier if $\mathrm{CO_2}$ is known since there is a high correlation (especially when EGR is activated), but NO_x models that do not need $\mathrm{CO_2}$ information are more useful as they can be applied without having to measure other emission values. Furthermore, we also excluded the variables $alpha$, COH, COL, THC, M_T01F, $ME\_MES01-07$, ME_MES10, ME_MES14 and ME_MES17 from the set of valid input variables for building models that do not incorporate exhaust information.

We applied GP using populations of 700 individuals for modeling the measured NO_x data; 1-elitism was applied, the mutation rate was set to 0.07, and rigid offspring selection was applied (maximum selection pressure: 300). The first 3,000 samples (representing 2.5 minutes) of the data sets were neglected; in strategy (1) the samples 3,001 – 10,000 were used as training data, in strategy (2) the samples 3,001 – 13,000. The rest of the data was used as validation / test samples.
Amongst other tests, we attacked modeling situation (1) without using exhaust information (hereafter called test strategy (1)), and modeling situation (2) using exhaust information (test strategy (2)); both test strategies were executed 5 times independently leading to the mean squared errors on training data summarized in Table 14.2.

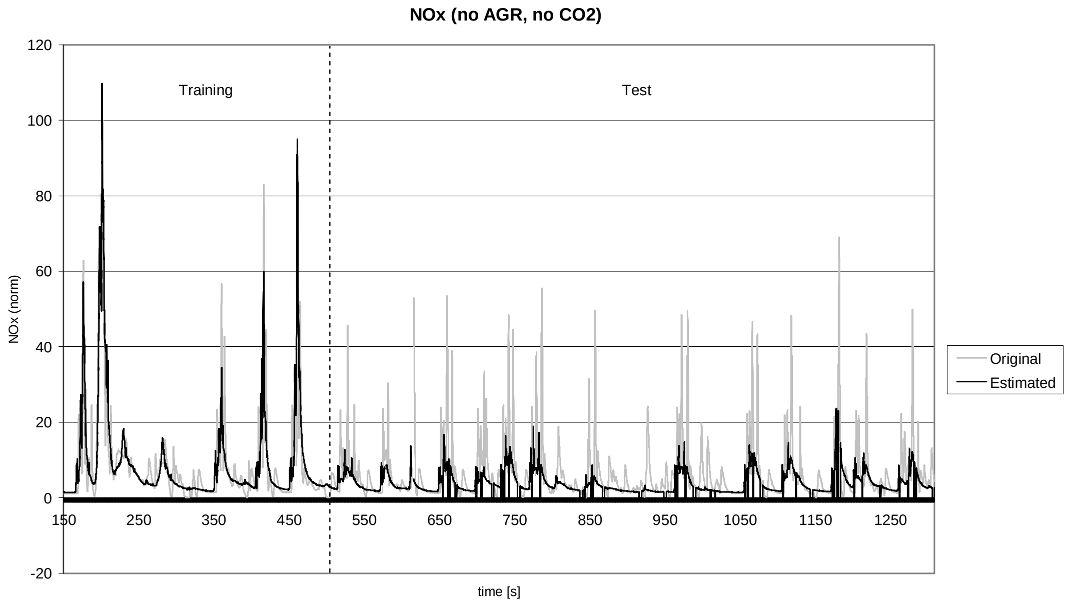
Let us now have a closer look at the best models (with respect to training data) produced for these test scenarios; their evaluations are both displayed in Figures 14.2

x_{i+2}; after sorting these values we get $x'_{i,j}$ for $j \in [1,5]$ with $x'_{i,j} < x'_{i,j+1}$ for $j \in [1,4]$. y_i is then set to the median value of x'_i, i.e. $y_i = x'_{i,3}$.

Table 14.2: Mean squared errors on training data for the NO_x data set.

	Test Strategy (1)	*Test Strategy (2)*
Average	49.867	13.454
Minimum	43.408	11.259
Maximum	58.126	18.432

and 14.3, respectively.

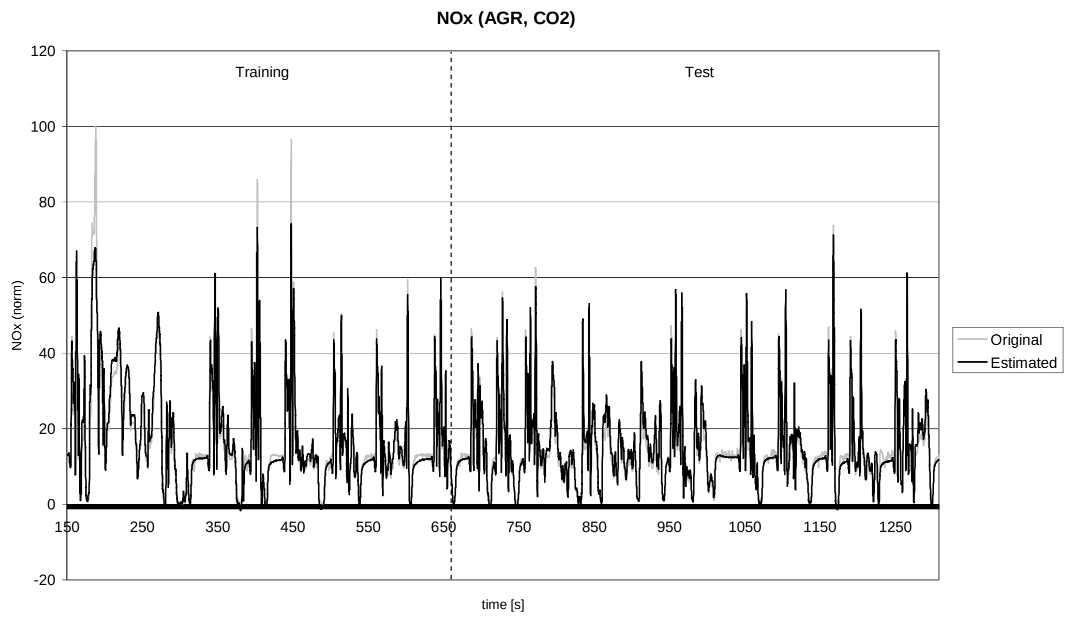


```
NOx_vK(t) = ((([0,950183*T_OEL(t-21)]^(([1,145998*T_LLVK(t-38)]-[1,023461*ME_MES17(t-28)])/([1,065444*ME_MES10(t-18)]+[0,951174*N_MOTOR(t-20)])))-(((([0,825300*
            ME_MES9(t-27)]+([0,871124*ME_MES9(t-22)]/[0,933514*ME_MES9(t-13)]))/([1,006088*N_MOTOR(t-21)]/[0,991764*ME_MES9(t-26)]))*((([0,891293*
            ME_MES9(t-15)]/[0,705167*N_MOTOR(t-26)])/([0,947497*N_MOTOR(t-25)]/[1,136678*ME_MES11(t-8)]))/(([1,082096*N_MOTOR(t-1)]+[1,286062*T_LLVK(t-37)])/
            -9,654))))+((([1,205990*T_OEL(t-30)]+[0,937356*T_OEL(t-10)])^([1,065581*T_LLVK(t-40)]/([0,928334*T_OEL(t-6)]+[1,184806*ME_MES7(t-12)])))-([0,888947*
            ME_MES9(t-14)]*(((([1,181614*ME_MES9(t-28)]/[0,666162*N_MOTOR(t-25)])/([0,631016*N_MOTOR(t-24)]/[1,301366*ME_MES11(t)]))/(([0,867211*
            N_MOTOR(t-40)]+[1,049105*T_LLVK(t-34)])/-7,628)))))
```

Figure 14.2: Evaluation of the best model produced by GP for test strategy (1).

The best model for test strategy (1) has a worse fit on test data ($mse_{test}(best_1) = 60.636$ opposed to $mse_{training}(best_1) = 43.408$); the best model for test strategy (2) surprisingly even has a better fit on test data ($mse_{test}(best_2) = 5.809$) than on training data ($mse_{train}(best_2) = 11.259$).

These research results were indeed promising, but still not completely satisfactory; in fact, these results started a series of data based approaches using GP in the context of mechatronical systems. The design of virtual sensors for NO$_x$ is to be again discussed in Chapter 16 and in Chapter 18 on the incorporation of physical knowledge about the formation of NO$_x$ in the GP process.



NOx_vK(t) = ((((\[0,838023*CO2_vK(t-6)]-(\[0,985178*T_LLVK(t-3)]/9,329))+((\[1,096131*ME_MES13(t-14)]-9,835)/(\[1,075515*ME_MES13(t-2)]+1,117)))+(((9,269-\[1,034379*THC_vK(t-8)])-(\[1,033333*T_ABGAS(t-13)]+\[COL_vK(t-17)]))/((-7,819+\[0,908894*N_MOTOR(t-14)])*\[1,216018*ME_MES13(t-15)])))-(((\[0,768676*CO2_vK(t-10)]-\[0,899741*ME_MES13(t-3)])/(((\[1,089508*ME_MES3(t-9)]/(\[1,200488*N_MOTOR(t-9)]*\[1,150031*ME_MES9(t-6)]))+(\[0,724729*CO2_vK(t-4)]+-1,733))+8,329)))

Figure 14.3: Evaluation of the best model produced by GP for test strategy (2).

We also tested standard GP without offspring selection, but with proportional as well as tournament ($k = 3$) parents selection, 1000 individuals, 2000 iterations, 7% mutation rate and the same data base as the one described previously.
Especially the use of proportional selection did not yield reasonable results, the evaluation of the best model for test strategy (1) returned mean squared error 110.23 on training data, and for the best for test strategy (2) the mean squared error was 21.34. The results obtained using tournament selection, which is anyway suggested in GP literature (as for example in [KKS$^+$03b] or [LP02]), were a lot better, but still not as good as those produced by extended GP: The best model for test strategy (1) showed mean squared error 61.92 on training data, and the best for test strategy (2) showed mean squared error 14.33. These results were no surprise, especially as we had seen on synthetic data sets that GP using rigid OS and gender specific parents selection performs a lot better than standard GP ([WAW04b], [Win04]).

All these results encouraged us to enforce research on the use of extended GP in the identification of mechatronical systems.

14.1.2 Designing Virtual Sensors for Particulate Emissions (Soot)

A lot of research work was done by DesCon members on the identification of particulate emissions of a BMW diesel engine. The main results have been published in [AdRWL05] and [LAWdR05], we shall here only summarize these results in a rather compact way.

In short, first attempts to use GP for producing models for soot were not very successful; GP did not produce any useful solution without restriction of the search space. Therefore, a two step approach was used: "In a first step, a statistical analysis was done on the basis of steady state measurements. Expert knowledge was combined with statistical correlations to yield an accurate steady state model. The advantage of steady state analysis is the secure validation of the model; any delay time or sensor dynamics are irrelevant. However, such a model could never meet the requirements of estimating the highly dynamical process of an IC engine. Therefore the steady state model is used as origin for the genetic programming cycle." (Copied from [AdRWL05] where this static model is given in detail.)

Using this static model, an additional variable was calculated and inserted into the set of potential input variables; this so enhanced variables set was then used as basis for data based identification of soot. In fact, this approach is equivalent to one of the possibilities for including a priori knowledge into the GP process as described in Section 9.

This extended data basis was used by two modeling approaches, namely a neural network training algorithm as well as GP; the best results for the ANN approach were achieved using a network structure with 2 hidden layers and 25 hidden nodes per layer, the parameters of the GP based training algorithm were set to our standard GP settings (1000 individuals, 10% mutation rate, rigid OS, 1-elitism). Again, the data were measured during a standard FTP engine test lasting approximately 23 minutes; the first approximately 8 minutes were taken as training, the rest as validation / test data set.

Figure 14.4 shows a detail of the evaluation of the models produced by GP and ANN on validation data: As we see clearly, both virtual sensors do not capture the behavior completely correctly, but the GP model's fit seems to be better than the one of the ANN model. This suspicion becomes clearer by analyzing the distribution of errors which is shown in Figure 14.5: The errors caused by the evaluation of the model produced by GP are more symmetric than those of the ANN[3] which can be considered an indication for a rather good model. The cumulative errors of these models are shown in 14.6, and we here see that the model produced by GP is able to reproduce the engine's cumulated soot emissions quite well.

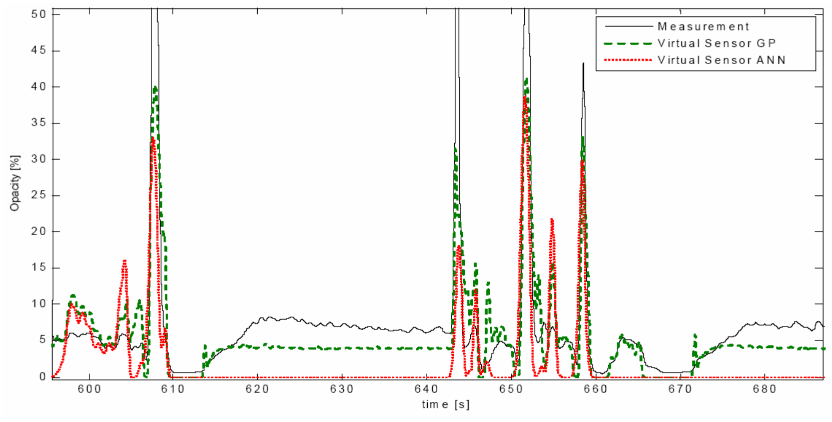


Figure 14.4: Evaluation of models for particulate matter emissions of a diesel engine (snapshot of the evaluation on validation / test samples), copied from [AdRWL05].

Again, these results were by far not completely satisfactory; of course, the ANN model could be improved by changing the network structure or the number of training iterations, and the GP process was not enhanced with local optimization or pruning operations. Still, again, these results sustained our confidence in GP's ability to produce reasonable models for mechatronical systems.

14.2 NO_x Data Sets Used for Further Tests

The NO_x data set described in Section 14.1 was used for several research activities of DesCon members as well as in our project investigating GP for the design of virtual sensors. Nevertheless, during the execution of our research project several other measurements were recorded and used for research; two of them were also used for test series that will be reported on in the following chapters. This is why we describe and characterize these data set here in Sections 14.2.1 and 14.2.2.

[3]In addition to GP and ANN, an auto-regressive moving-average with exogenous inputs (ARMAX) modeling approach was also calculated for reasons of comparison; the distribution of the errors caused by the evaluation of this model are also shown in Figure 14.5. Please see [BJ76] for explanations and application examples of ARMA(X) models.

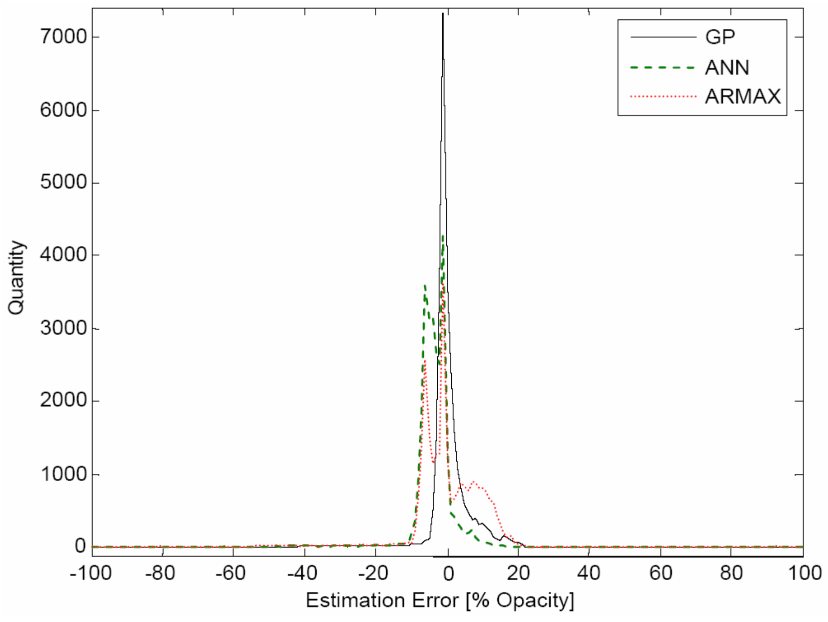


Figure 14.5: Distribution of errors caused by models for particulate matter emissions, copied from [AdRWL05].

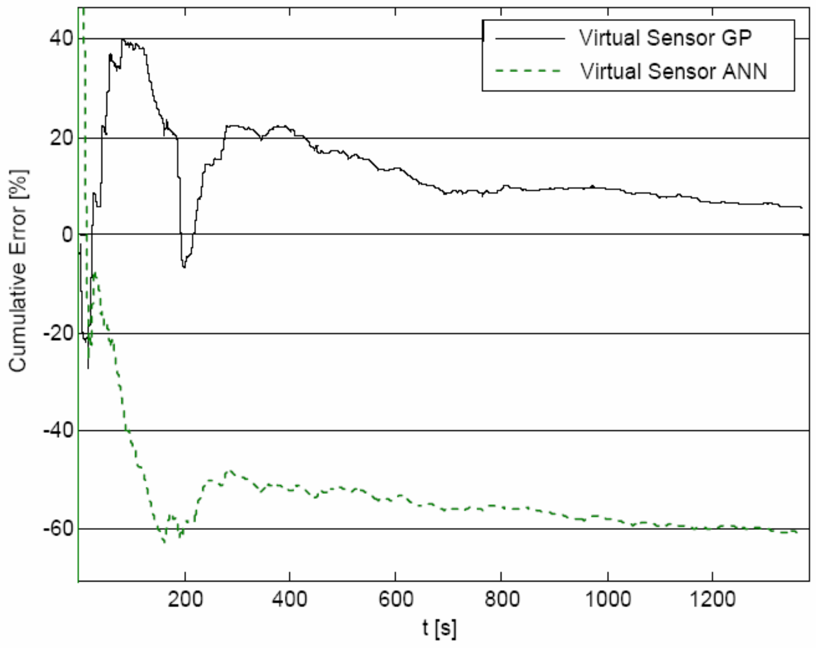


Figure 14.6: Cumulative errors caused by models for particulate matter emissions, copied from [AdRWL05].

14.2.1 NO_x Data Set II

Recorded in 2006 by members of the Institute for Design and Control of Mechatronical Systems at JKU Linz at the test bench already mentioned, this NO_x data set includes the variables listed in Table 14.3. The data set available in this context again contains measurements taken from a 2 liter 4 cylinder BMW diesel engine. Again, several emissions (including NO_x, CO and CO_2) as well as several other engine parameters were recorded at 100 Hz and downsampled to 20 Hz. 22 signals were recorded over approximately 18 minutes, but only 9 variables were considered in further identification test series.

Several variables were measured over approximately 30 minutes at 100 Hz recording frequency; they have been downsampled to 20 Hz, so that the resulting data set includes ∼36,000 samples. From the variables recorded several have been removed (as for example CO, CO_2 and *time*) due to irrelevance or high correlations with the target variable Nox_true; the 10 remaining variables are characterized in Table 14.3, Figure 14.7 shows a graphical representation of the target values over the whole recording time.
The variable NOx_Can represents values given by a quick, but also rather imprecise estimation for the NO_x emissions; the actual NO_x emissions were again measured using a Horiba Mexa 7000 combustion analyzer, the respective values are stored in variable Nox_true.

Table 14.3: Statistic features of the identification relevant variables in the NOx data set II.

Variable	*Minimum*	*Maximum*	*Mean*	*Variance*
(0) Eng_nAvg	0.00	3,311.00	1,618.80	413,531.96
(1) $AFSCD\_mAirPerCyl$	-44.56	1,161.36	453.12	60,952.03
(2) $VSACD\_rOut$	5.00	96.00	33.59	1,706.83
(3) NOx_CAN	-0.30	6.72	1.52	2.87
(4) T_OEL	78.68	100.83	87.57	31.05
(5) (T) Nox_true	62,46	1,115.23	225.25	60,673.98
(6) $InjCrv\_qPil1Des$	0.00	1.40	0.88	0.10
(7) $InjCrv\_qMI1Des$	0.00	57.93	12.63	122.73
(8) $InjCrv\_phiMI1Des$	-3.86	10.61	2.80	18.70
(9) $BPSCD\_pFltVal$	986.20	2,318.00	1214.89	104,434.00

All pairwise linear correlations[4] are summarized in Table 14.4; again, all vari-

[4]We here use the same standard formula for calculating linear correlation coefficients of time

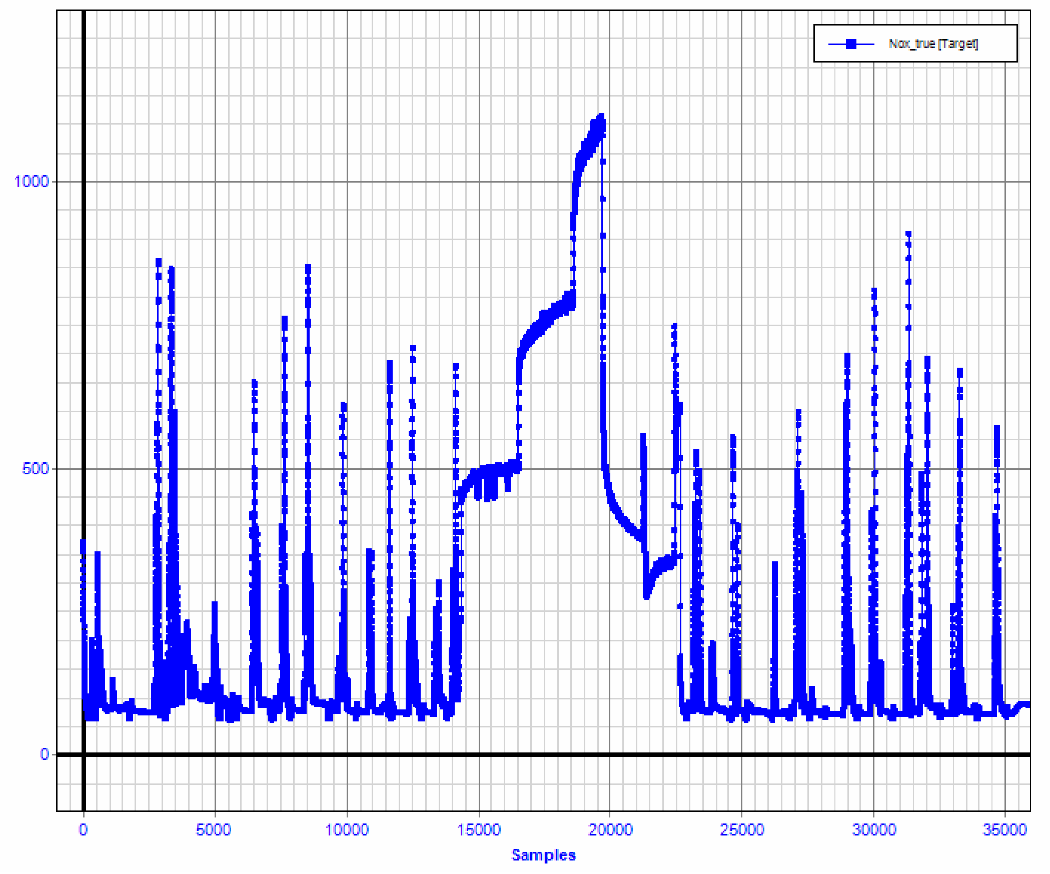


Figure 14.7: Target NO_x values of NOx data set II, recorded over approximately 30 minutes at 20Hz recording frequency yielding ~36,000 samples.

ables were filtered using a median filter of order 5 before calculating the correlation coefficients. Obviously, there is a rather high linear correlation between the target variable and the input variables $BPSCD\_pFltVal$ and NOx_CAN; the values stored in $AFSCD\_mAirPerCyl$ and $InjCrv\_qMI1Des$ are also remarkably correlated to the designated target values.

14.2.2 NO_x Data Set III

During the time in which we were doing the research work reported on in this thesis, maintenance work was repeatedly done at the DesCon test bench; amongst other aspects, several sensors were removed or replaced by newer ones.

series as described in Section 11.1.

Table 14.4: Linear correlation coefficients of the variables relevant in the NOx identification task II.

	(0)	(1)	(2)	(3)	(4)	NOx	(6)	(7)	(8)	(9)
(0) Eng_nAvg	1.00	0.80	0.52	0.70	0.61	0.65	0.59	0.65	0.68	0.75
(1) $AFSCD\_mAirPerCyl$	0.80	1.00	0.78	0.90	0.80	0.91	0.60	0.88	0.63	0.95
(2) $VSACD\_rOut$	0.53	0.78	1.00	0.73	0.77	0.78	0.38	0.77	0.63	0.81
(3) NOx_CAN	0.70	0.90	0.73	1.00	0.74	0.93	0.63	0.86	0.58	0.94
(4) T_OEL	0.61	0.80	0.77	0.74	1.00	0.78	0.51	0.75	0.49	0.81
(5) (T) NOx_true	0.65	0.91	0.78	0.93	0.78	1.00	0.61	0.90	0.60	0.95
(6) $InjCrv\_qPil1Des$	0.59	0.60	0.38	0.63	0.51	0.61	1.00	0.70	0.03	0.62
(7) $InjCrv\_qMI1Des$	0.65	0.88	0.77	0.86	0.75	0.90	0.70	1.00	0.50	0.87
(8) $InjCrv\_phiMI1Des$	0.68	0.63	0.63	0.58	0.49	0.60	0.03	0.50	1.00	0.66
(9) $BPSCD\_pFltVal$	0.75	0.95	0.81	0.94	0.81	0.95	0.61	0.87	0.66	1.00

The third NO_x data set was recorded in 2007 by members of DesCon; again, several variables were measured at the motor engine test bench while testing a 2 liter 4 cylinder BMW diesel engine (simulated vehicle: BMW 320d Sedan). The mean engine speed was set to 2,200 revolutions per minute (rpm), and in each engine cycle 15mg fuel were injected.
Once again, several emissions (including NO_x, CO and CO_2) as well as several other engine parameters were recorded; this time the measurements were recorded over approximately 18.3 minutes at 100 Hz and then downsampled to 10 Hz, yielding a data set containing ~11,000 samples. The target values (the engine's NO_x emissions measured by a Horiba combustion analyzer) are stored in variable $HoribaNOx$.

In Chapter 18 we report on tests in which we have used this data set for testing the ability of GP to incorporate physical knowledge. For this purpose we also use a synthetic variable HFM^*:

$$HFM^* = \frac{HFM}{N} \cdot \frac{1000}{60} \tag{14.1}$$

This synthetic variable is also included in this NO_x data set III; detailed explanations regarding the meaning of this additional variable can be found in Chapter 18.

Figure 14.8 visualizes all target $HoribaNOx$ values available (in total approximately 11,000 samples); Figure 14.9 shows a detail of these data, namely the $HoribaNOx$ of samples 6000 – 7000.

In detail, Table 14.5 summarizes the main statistic parameters of the variables relevant in this identification task. Again, all pairwise linear correlations have also been calculated, the results are summarized in Table 14.6; all variables were again filtered using a median filter of order 5 before calculating the correlation coefficients. As we see in this table, there are no remarkably high correlations except for the obvious one between HFM and HFM^*; the correlation coefficient of HFM^* and the target, $HoribaNOx$, is above average (0.72), but not high enough to build a

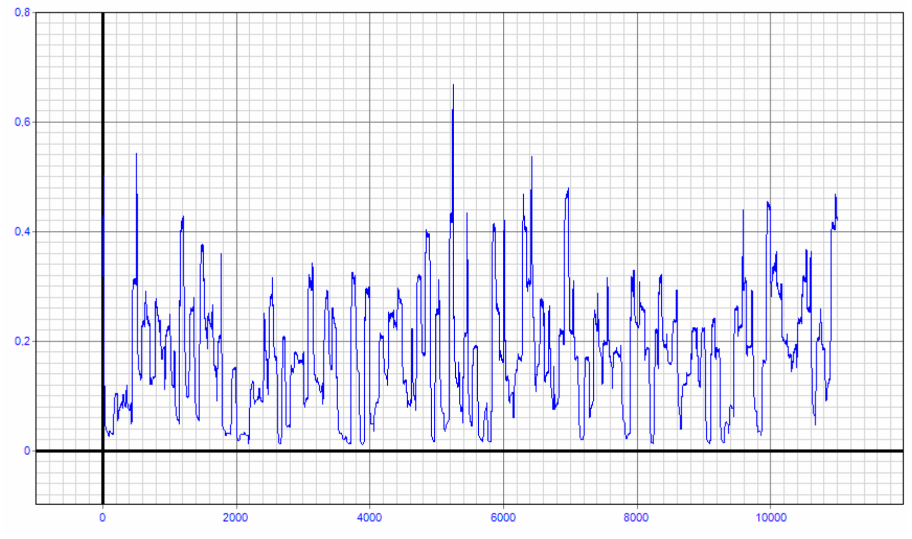


Figure 14.8: Target $HoribaNOx$ values of NO_x data set III.

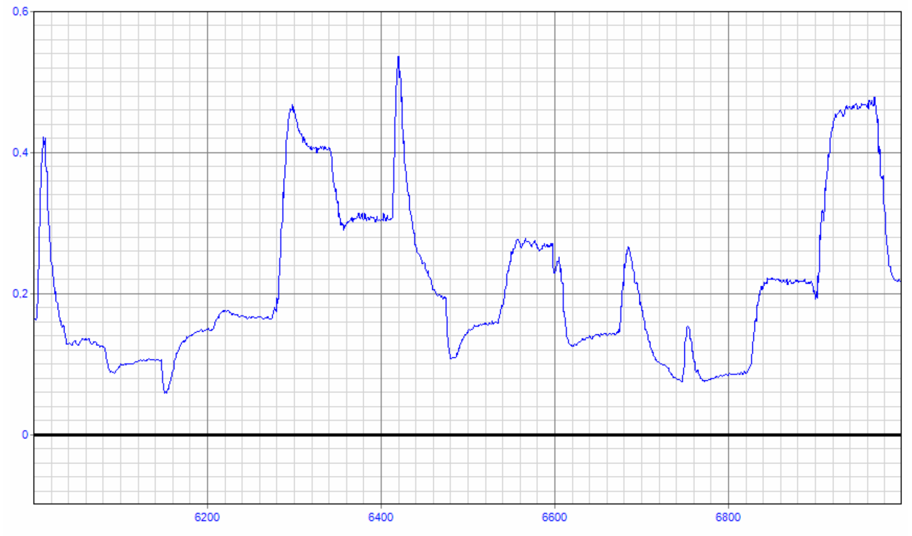


Figure 14.9: Target $HoribaNOx$ values of NO_x data set III, samples 6000 – 7000.

reasonable model only using this variable as input.

The results of the correlation analysis is also graphically displayed in Figure 14.10(a): Each correlation coefficient for variables Var_i and Var_k is represented in cell (i, k), where lower values are represented by blue cells and red cells indicate high correlation values.
As there could be correlations between variables and time-delayed values of other variables, we additionally collected all variables with time delays in

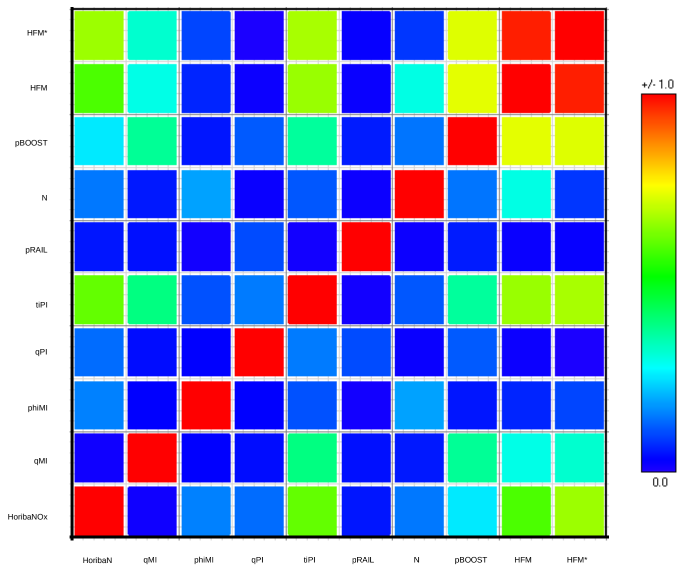
Table 14.5: Statistic features of the identification relevant variables in the NOx data set III.

Variable	*Minimum*	*Maximum*	*Mean*	*Variance*
(0) (T) $HoribaNOx$	0.011	0.670	0.171	0.011
(1) qMI	8.010	21.960	15.232	16.992
(2) pMI	-0.727	8.016	3.424	6.525
(3) qPI	0.000	2.480	0.929	0.627
(4) $tiPI$	0.018	6.690	4.425	1.358
(5) $pRAIL$	487.900	927.400	709.355	13,334.040
(6) N	1,906.000	2,507.000	2,208.384	27,668.381
(7) $pBOOST$	981.000	1906.000	1209.841	28,618.435
(8) HFM	15.148	241.628	101.290	1,226.203
(9) HFM^*	0.105	1.627	0.763	0.062

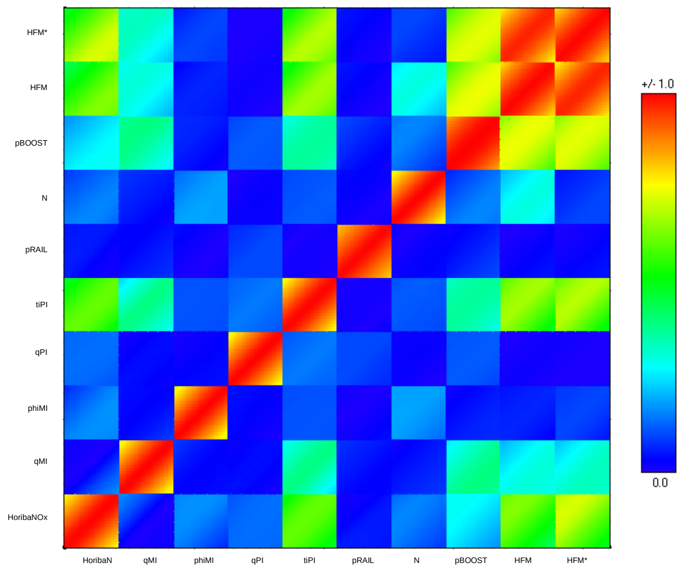
Table 14.6: Linear correlation coefficients of the variables relevant in the NOx identification task III.

	$HoribaNOx$	qMI	pMI	qPI	$tiPI$	$pRAIL$	N	$pBOOST$	HFM	HFM^*
(0) (T) $HoribaNOx$	1.00	0.01	0.15	-0.13	0.61	-0.05	-0.14	0.25	0.59	0.67
(1) qMI	0.01	1.00	0.03	0.04	-0.39	-0.04	-0.05	0.37	0.29	0.32
(2) pMI	0,15	0.03	1.00	-0.03	-0.11	0.01	0.18	-0.05	-0.06	-0.10
(3) qPI	-0.13	0.04	-0.03	1.00	-0.14	-0.10	0.02	0.11	0.01	0.00
(4) $tiPI$	0,61	-0.39	-0.10	-0.14	1.00	-0.01	0.11	0.37	0.66	0.68
(5) $pRAIL$	-0.05	-0.04	0.01	-0.10	-0.01	1.00	-0.02	-0.05	-0.02	-0.02
(6) N	-0.14	-0.05	0.18	0.02	0.11	-0.02	1.00	0.14	0.30	0.08
(7) $pBOOST$	0.25	0.37	-0.05	0.11	0.37	-0.05	0.14	1.00	0.73	0.73
(8) HFM	0.59	0.29	-0.06	0.01	0.66	-0.02	0.30	0.73	1.00	0.97
(9) HFM^*	0.67	0.32	-0.10	0.00	0.68	-0.02	0.08	0.73	0.97	1.00

the range $[1;20]$; thus, what we get is a temporary set of variables $X = \{V0_{t-20}, V0_{t-19}, \ldots, V0_t, V1_{t-20}, \ldots, V1_t, \ldots, V9_{t-20}, \ldots, V9_t\}$ where N is the number of variables (in our case 10) and $Vj_{t-k}(i) = Vj(i-k)$ for $j \in [0;9]$. For all these (temporary) time shifted variables we calculate a full linear correlation analysis. The results are graphically displayed in Figure 14.10(b), where the correlation of (temporary) variables Xi and Xj is shown in cell (i,j) $(i,j \in [0;210])$; obviously, there are no more remarkable correlations additional to those already mentioned before.



(a) Correlations of variables included in the NO_x data set III.



(b) Correlations of variables (delayed up to 20 samples) included in the NO_x data set III.

Figure 14.10: Correlations of variables included in the NO_x data set III.

14.3 Modeling High Pressure Differences in a Tractor Gearbox

In this section we discuss the test results obtained analyzing data provided by *Hofer Forschungs- und Entwicklungs-GmbH & CoKG* [5] at Garsten, Upper Austria. We are especially thankful to Ing. Reinhard Flachs, Dipl.-Ing. Heinz Aizetmüller and Dipl.-Ing. Roland Gerbis who provided the data base and gave important hints during discussions about the data.

14.3.1 The *Gearbox* Data Set

The data set provided by Hofer contains data recorded during a test run of a traction engine; the data contains gearbox signals, one of them being the "Hochdruckdifferenz" (difference of high pressure values, hereafter denoted as the target variable $Var10$) which is the target variable for future modeling experiments. 20,001 samples of 11 variables are included in this data set (hereafter referred to as the *Gearbox* data set), the data were recorded at 100 Hz sampling frequency.

In detail, Table 14.7 summarizes the main statistic parameters of the variables relevant in this identification task. All pairwise linear correlations[6] are summarized in Table 14.8; again, all variables were filtered using a median filter of order 5 before calculating the correlation coefficients. As we see in this table, there are in some cases high correlations, but none of the potential input variables has a strong (linear) correlation to the target variable. Figure 14.11 graphically shows this correlation analysis by displaying a color representation of the correlation between each pair of signals i and j in cell (i, j) ($i \in [0; 10]$ and $j \in [0; 10]$).

14.3.2 Modeling Methods Used for Analyzing the *Gearbox* Data Set

In the context of larger data analysis projects and automated data analysis processes, there are normally several data processing steps that are to be executed

[5]The webpage of *Hofer Powertrain GmbH* can be found at `http://www.hofer.de/`, detailed information about the company's subsidiary at Garsten, Steyr on `http://www.hofer.de/de/Kontakt_Standorte_Steyr.html`.

[6]We here use the same standard formula for calculating linear correlation coefficients of time series as described in Section 11.1.

Table 14.7: Statistic features of the relevant variables in the *Gearbox* data set.

Variable	*Minimum*	*Maximum*	*Mean*	*Variance*
$Var00$ (time)	0.010	199.990	100.000	3,333.833
$Var01$	-0.893	0.999	0.363	0.243
$Var02$	-0.944	1.038	0.346	0.250
$Var03$	-3,278.000	2,700.250	-912.481	1,916,915.678
$Var04$	28.750	35.250	32.747	1.115
$Var05$	48.750	2,855.500	1,622.801	361,949.240
$Var06$	0.000	5,279.250	2,167.383	2,186,999.143
$Var07$	0.000	2,075.999	1,344.028	414,557.048
$Var08$	0.000	2,898.500	1,866.742	799,737.150
$Var09$	875.250	2,387.500	1,841.324	59,035.060
$Var10$ (target)	-88.078	300.266	98.838	2,029.311

Table 14.8: Linear correlation coefficients of relevant variables in the *Gearbox* identification task.

	$Var00$	$Var01$	$Var02$	$Var03$	$Var04$	$Var05$	$Var06$	$Var07$	$Var08$	$Var09$	$Var10$
$Var00$	1.00	-0.44	-0.45	0.37	0.32	0.45	0.27	0.61	0.61	0.48	0.21
$Var01$	-0.44	1.00	1.00	-0.99	0.00	0.02	-0.61	-0.50	-0.50	-0.13	-0.07
$Var02$	-0.45	1.00	1.00	-0.99	-0.01	0.02	-0.61	-0.50	-0.50	-0.15	-0.09
$Var03$	0.37	-0.99	-0.99	1.00	-0.12	-0.10	0.59	0.44	0.44	0.01	0.02
$Var04$	0.32	0.00	-0.01	-0.12	1.00	0.54	0.20	0.43	0.43	0.95	0.48
$Var05$	0.45	0.02	0.02	-0.10	0.54	1.00	0.03	0.85	0.85	0.52	-0.13
$Var06$	0.27	-0.61	-0.61	0.59	0.20	0.03	1.00	0.34	0.34	0.25	0.12
$Var07$	0.61	-0.50	-0.50	0.44	0.43	0.85	0.34	1.00	1.00	0.48	-0.11
$Var08$	0.61	-0.50	-0.50	0.44	0.43	0.85	0.34	1.00	1.00	0.48	-0.11
$Var09$	0.48	-0.13	-0.15	0.01	0.95	0.52	0.25	0.48	0.48	1.00	0.53
$Var10$	0.21	-0.07	-0.09	0.02	0.48	-0.13	0.12	-0.11	-0.11	0.53	1.00

before applying modeling methods. E.g., data downsampling, correlation analysis and variable selection methods are some of the most common preprocessing steps; modeling is usually done on basis of the data produced by these preprocessing steps. Here, indeed, we have intentionally restricted the data analysis process to merely splitting the data into a training- and a test-data set (the first 10,000 samples are used as training data) and training models using these test samples. The following modeling methods have been tested: Linear regression modeling (Lin), artificial neural networks (ANNs) and genetic programming (GP).

Since it seems to be useful to create models that use information about the signals' past values for estimating the target values, an extended data set has also been created in order to be able to use also other modeling approaches than GP; this extended data base includes the same signals, but also duplicates shifted by 1,

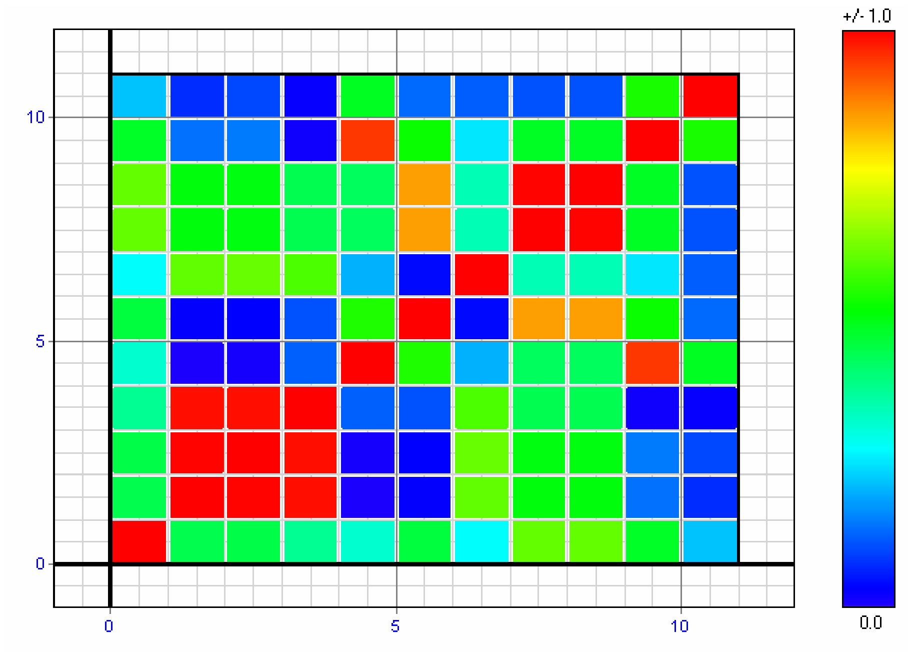


Figure 14.11: Graphical display of linear correlations among variables included in the *Gearbox* data set.

2, . . . and 20 samples. This of course has the effect that the number of potential input signals is multiplied by 21 when using this extended data set.

14.3.2.1 Linear Modeling

Given a data collection including m input features storing the information about N samples, a linear model is defined by the vector of coefficients $\theta_{1...m}$. For calculating the vector of modeled values e using the given input values matrix $u_{1...m}$, these input values are multiplied with the corresponding coefficients and added:

$$e = u_{1...m} * \theta \tag{14.2}$$

The coefficients vector can be computed by simply applying matrix division. For conducting the test series documented here we have used the matrix division function provided by MATLAB©:

```
theta = InputValues \ TargetValues;                (14.3)
```

If a constant additive factor is to be included into the model (i.e., the coefficients vector), this command has to be extended:

```
r = size(InputValues,1);                                      (14.4)
theta = [InputValues ones(r,1)] \ TargetValues;               (14.5)
```

Theoretical background of this approach can be found in [Lju99].

14.3.2.2 Neural Networks

For training artificial neural network (ANN) models, three-layer feedforward neural networks with one output neuron were created using the Levenberg-Marquardt training method. Theoretical background and details can be found in [Nel01] (Chapter 11, "Neural Networks"), [Mar63], [Lev44] or [GMW82] ("The Levenberg-Marquardt method", pp. 136–137). The ANN training framework used to collect the results reported here is the NNSYSID20 package, a neural network toolbox for MATLAB© implemented by Magnus Nørgaard at the Technical University of Denmark [Nør00].

14.3.2.3 Genetic Programming Based Structure Identification

Finally, we also used GP based structure identification as described in the first part of this thesis for identifying a model for the given training data. Gender specific selection and strict offspring selection (as described in Sections 4.1 and 4.2) as well as additional optimization stages (including periodical pruning and parameters optimization applied to 20% of the population in each 5^{th} iteration) have been applied.

14.3.3 Test Results

Each modeling method (except linear modeling which does not include any stochastic elements) was executed several times, also trying different sets of parameter settings. All constellations of algorithmic approaches and parameter settings were executed independently at least 3 times; we shall here report on the best results (with respect to modeling quality on training data) obtained for each constellation.

14.3.3.1 Linear Modeling

A linear model obtained was calculated in MATLAB© using the first 10,000 samples as training data, the result being the parameters vector θ:

$$\theta = [1.30;\ 379.04;\ -324.08;\ -0.09;\ -1.56;\ -0.49;\ 0.004;\ 0.14;\ 0.21;\ 0.13;\ -63.33] \tag{14.6}$$

As we had expected, the result is not satisfying since the model does not seem to reproduce the target data properly. This can be seen in the mean squared errors (MSE):
$MSE_{training}(lin1) = 911.99$, $MSE_{test}(lin1) = 15,947.19$

Since standard linear regression is not able to consider a history of the given signals, the linear modeling process has also been tested on the extended data set (of course producing a much bigger model, it is therefore not stated here). Obviously, here the result is better than before (at least on training data, but not on the test samples set), but still not satisfying since the model does not seem to reproduce the target data properly. This can be also seen in the mean squared errors:
$MSE_{training}(lin2) = 369.11$, $MSE_{test}(lin2) = 112,135.94$

In Figure 14.12 we graphically show the full original data as well as predicted target data (the first 10,000 samples being training, the rest test data) using the model built on the extended data set.

14.3.3.2 Neural Networks

There are several parameters that can influence the neural network produced by a NN-based modeling process; the most important ones are the number of hidden nodes (NHN) and the number of iterations (IT) (we have restricted our experiments to network structures working with one layer of hidden nodes). Increasing the number of nodes and iterations will in most cases lead to models that perform better on training data, but might also lead to overfitting and so to models that perform worse on test data. So we have tried several different parameter settings and document the results of the respective experiments in Table 14.9.

Since standard neural network implementations are also not able to consider a history of the given signals, the NN-based modeling process has also been tested on the extended data set. Again, we have tried several different parameter settings and document the results of the respective experiments:

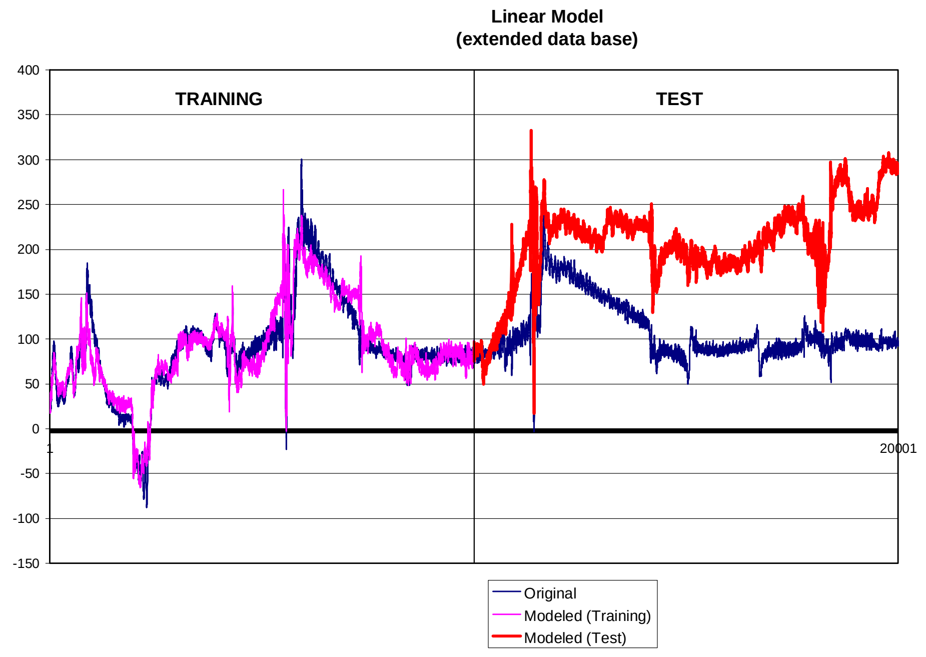


Figure 14.12: Graphical display of the evaluation of a linear model calculated for the extended *Gearbox* data set.

Table 14.9: Network parameters and evaluation of the best NN models for the *Gearbox* data set.

NN Variant	*NHN*	*IT*	$MSE_{training}$	MSE_{test}
(1)	1	200	856.08	7,149.86
(2)	2	800	405.45	2,777.45
(3)	3	1000	348.81	12,936.49
(4)	5	2000	184.99	2,277.82

14.3.3.3 Genetic Programming Based Structure Identification

Finally, we have tested advanced GP-based modeling using the parameter settings summarized in Table 14.11.

Especially with respect to the performance on test data, it has to be stated that the results obtained using GP are obviously the best of all models created during this data analysis studies. Even though NNs were able to train models that perform a lot better on training data, the formulae produced by GP show a significantly lower error on test data samples.

Table 14.10: Network parameters and evaluation of the best NN models for the *Gearbox* data set.

NN Variant	*NHN*	*IT*	$MSE_{training}$	MSE_{test}
(5)	1	50	375.93	10,567.40
(6)	1	200	453.46	1,085.67
(7)	3	50	98.79	2,402.16
(8)	3	500	189.66	3,120.62
(9)	5	50	100.67	2,973.30
(10)	5	1000	49.09	6,141.64
(11)	8	50	237.24	4,977.87
(12)	8	200	34.74	718.35
(13)	8	500	30.49	1,127.74

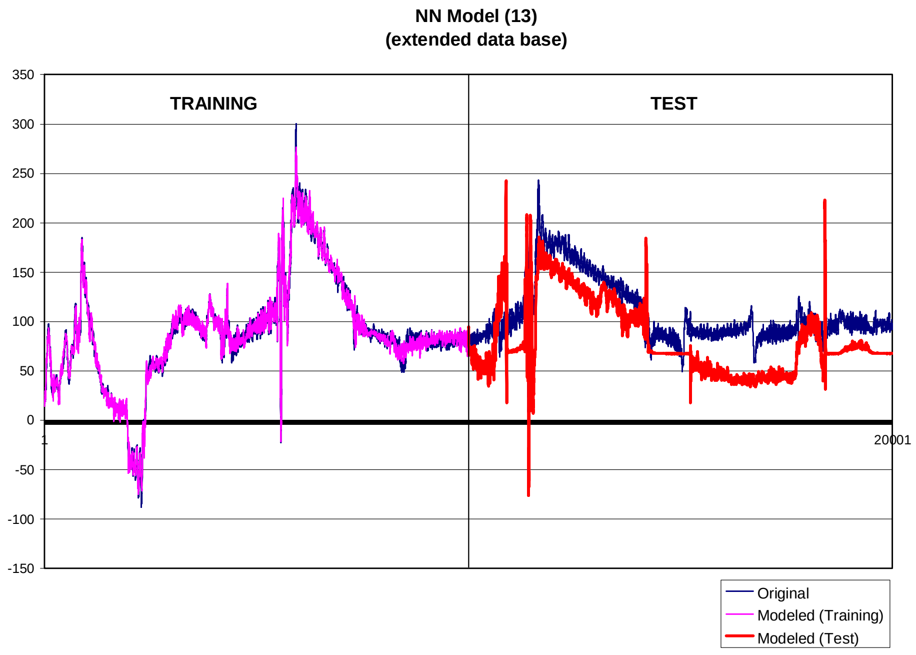


Figure 14.13: Graphical display of the evaluation of the NN model (13) calculated for the extended *Gearbox* data set.

The best model produced by GP (with respect to training data fit) is shown in Figure 14.14, a graphical representation of this model's evaluation is given in Figure 14.15.

In Figure 14.16 we give comparative charts displaying the original target values vs. the estimated values: Each sample is represented by one point in the chart; each

Table 14.11: Algorithmic parameters and evaluation results for enhanced GP based modeling applied to the *Gearbox* data set.

Parameter	*Status*	*Notes*
Population Size	1000	
Mutation Rate	0.15	Parametric as well as structural
Parental Selection	Gender Specific	Random & proportional
Offspring Selection	Applied	Success Ratio: 1.0 Maximum Selection Pressure: 100
Elitism	1-Elitism	
Max. Time Offset	20	
Max. Tree Height	8	
Max. Tree Size	255	
Max. Time Offset	20	
Pruning	Applied	Full (1+λ)-ES based pruning after each 7th generation Maximum deterioration: 1.5 Maximum deterioration coefficient: 1.0 α: 10, maximum number of rounds: 50
Parameter Optimization	Applied	Full parameter optimization after each 5th generation λ: 10, maximum number of rounds: 50
Rounds	68 - 72	
Effort	$\sim$ 5,000,000 evaluations	Including evaluations during pruning and parameters optimization
MSE$_{training}$		
min	530.76	
max	610.98	
average	590.12	
MSE$_{test}$		
min	353.58	
max	631.53	
average	673.23	

point's x value is set to the respective sample's original target value, the y value to the respective sample's estimated target value. In the charts given in Figure 14.17 we finally show the error distribution of the result achieved using the GP machine learning approach (evaluated on training as well as on test data).

For the comparing these results with those produced by conventional GP we

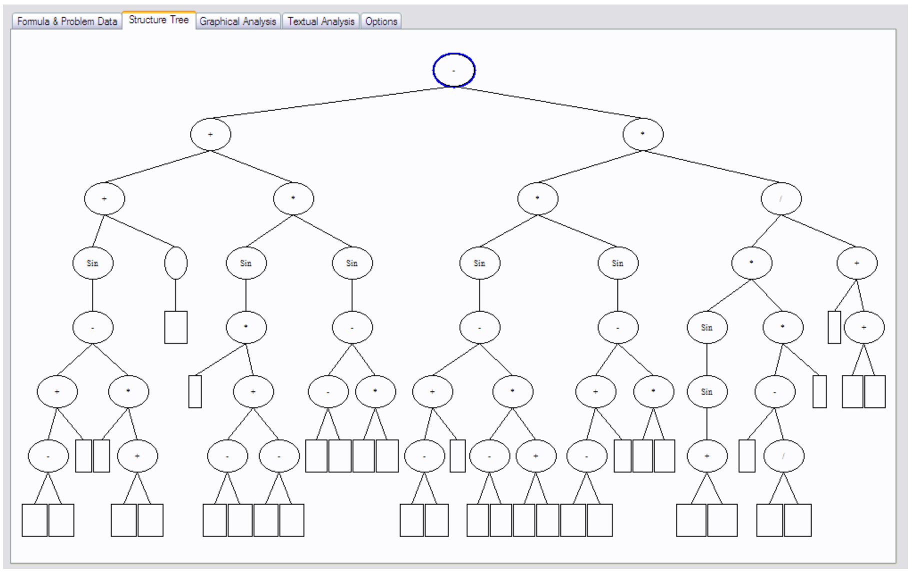


Figure 14.14: Graphical display of the best model produced by GP for the *Gearbox* data set.

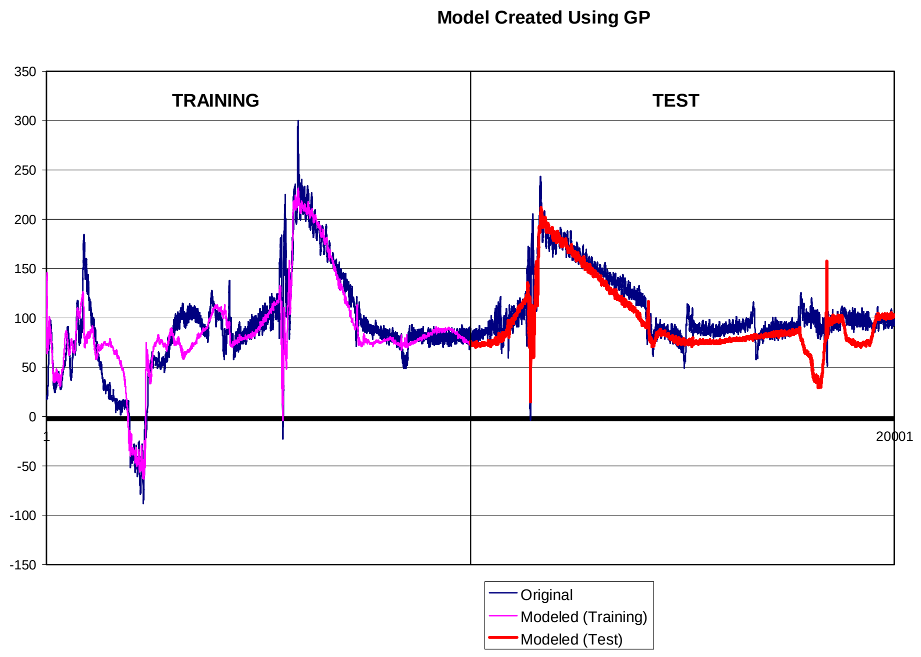


Figure 14.15: Evaluation of the best model produced by GP for the *Gearbox* data set.

have also tested standard GP using the settings summarized in Table 14.12; again, 5 independent test runs were executed. As we see in Table 14.12, the results using

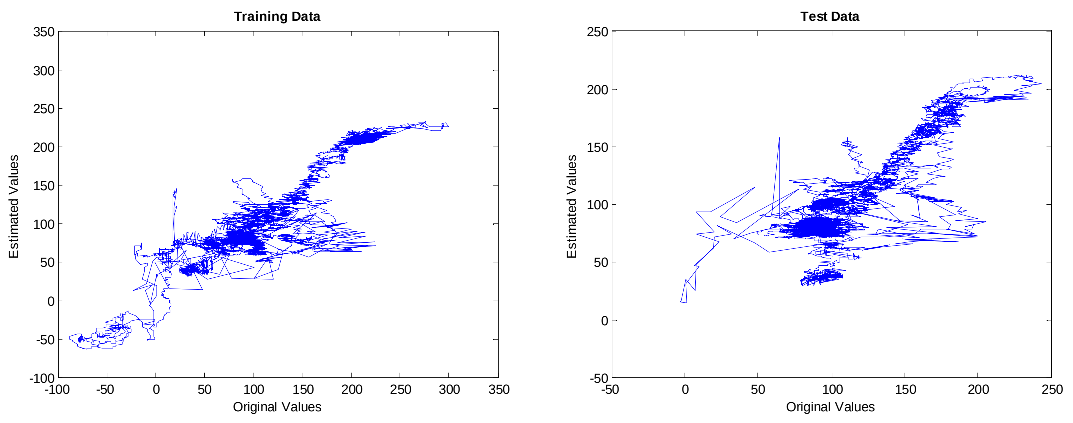


Figure 14.16: Original vs. estimated values, calculated using the evaluating best model for the *Gearbox* data set produced by GP. Training data are displayed in the left, test data in the right chart.

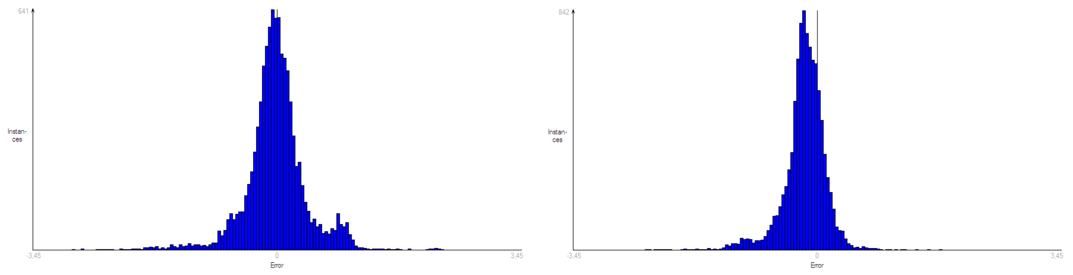


Figure 14.17: Error distributions of the best model produced for the *Gearbox* data set, evaluated on training and test data (shown in the left and the right part, respectively).

standard GP are comparable to those using extended GP, but not quite as good; in one case, a model with better fit on training data was produced, but the evaluation on test shows worse results for these models than those for extended GP stated in Table 14.11. Anyway, the result obtained using standard GP are still clearly better than those produced by NNs and linear modeling.

14.3.4 Conclusion

In this section we have described the results of system identification case study based on a data set provided by Hofer at Garsten, Upper Austria. We have applied several modeling methods, namely linear modeling, artificial neural networks and genetic programming; all methods were used for building dynamic models, i.e. formulas

Table 14.12: Algorithmic parameters and evaluation results for standard GP modeling applied to the *Gearbox* data set.

Parameter	*Status*	*Notes*
Population Size	2000	
Mutation Rate	0.15	Parametric and structural
Parental Selection	Tournament Selection ($k = 3$)	
Elitism	1-Elitism	
Max. Time Offset	20	
Max. Tree Height	8	
Max. Tree Size	255	
Max. Time Offset	20	
Rounds	3000	
Effort	6,000,000 evaluations	
$\text{MSE}_{training}$		
min	412.83	
max	730.12	
average	494.56	
MSE_{test}		
min	612.76	
max	737.22	
average	693.23	

that use past and current information of input variables for estimating the target variable's values.

As we have summarized in this section, linear modeling was not able to produce a satisfying model; neural networks perform a lot better and can be used for producing nearly optimal models for the given training data, i.e. models that reproduce the given data almost perfectly. Still, obviously these NN models tend to become worse on test data as they are more and more optimized to fit given training data.

Especially with respect to the performance on test data, the results obtained using enhanced genetic programming are obviously the best of all models created during this data analysis studies. Even though NNs were able to train models that perform a lot better on training data, the formulae produced by GP show a significantly lower error on test data samples.

Chapter 15

Classification

In this chapter we summarize the results of empirical studies in the context of solving various data based classification tasks. In Section 15.1 we summarize results of classification studies using medical benchmark data sets, and Section 15.2 summarizes research results in the context of quality pre-assessment in steel industry using data based estimators.

15.1 Medical Data Analysis

15.1.1 Benchmark Data Sets

For testing GP-based training of classifiers here we have picked the following data sets: The *Wisconsin Breast Cancer*, the *Melanoma* and the *Thyroid* data sets.

- The *Wisconsin* data set is a part of the UCI machine learning repository[1]. In short, it represents medical measurements which were recorded while investigating patients potentially suffering from breast cancer. The number of features recorded is 9 (all being continuous numeric ones); the file version we have used contains 683 recorded examples (by now, 699 examples are already available since the data base is updated regularly).
- The *Thyroid* data set represents medical measurements which were recorded while investigating patients potentially suffering from hypo- or hyperthy-

[1] `http://www.ics.uci.edu/~mlearn/`

roidism; this data set has also been taken from the UCI repository. In short, the task is to determine whether a patient is hypothyroid or not. Three classes are formed: Euthyroid (the state of having normal thyroid gland function), hyperthyroid (overactive thyroid) and hypothyroid (underactive thyroid).
In total, the data set contains 7200 samples. The samples of the *Thyroid* data set are not equally distributed to the three given classes; in fact, 166 samples belong to class "1" ("subnormal functioning"), 368 samples are classified as "2" ("hyperfunction"), and the remaining 6666 samples belong to class "3" ("euthyroid"); a good classifier therefore has to be able to correctly classify significantly more than 92% of the samples simply because 92 percent of the patients are not hypo- or hyperthyroid. 21 attributes (15 binary and 6 continuous ones) are stored in this data set.

- The *Melanoma* data set represents medical measurements which were recorded while investigating patients potentially suffering from skin cancer. It contains 1311 examples for which 30 features have been recorded; each of the 1311 samples represents a pigmented skin lesion which has to be classified as a melanoma or a nonhazardous nevus. This data set has been provided to us by Prof. Dr. Michael Binder from the Department of Dermatology at the Medical University Vienna, Austria.
  A comparison of machine learning methods for the diagnosis of pigmented skin lesions (i.e., detecting skin cancer based on the analysis of visual data) can be found in [DOMK$^+$01]; in this paper the authors describe the quality of classifiers produced for a comparable data collection using k-NN classification, ANNs, decision trees, and SVMs. The difference is that in the data collection used in [DOMK$^+$01] all lesions were separated into three classes (common nevi, dysplastic nevi, or melanoma); here we use data representing lesions that have been classified as benign or malign, i.e. we are facing a binary classification problem.

All three data sets were investigated via 10-fold cross-validation. This means that each original data set was divided into 10 disjoint sets of (approximately) equal size. Thus, 10 different pairs of training (90% of the data) and test data sets (10% of the data) can be formed and used for testing the classification algorithm.

The results summarized in this section have been partially published in [WAW06b], [WAW06e] and [WAW07a].

Table 15.1: Set of function and terminal definitions for enhanced GP based classification.

Functions		
Name	**Arity**	**Description**
$+$	2	Addition
$*$	2	Multiplication
-	2	Subtraction
/	2	Division
e^x	1	Exponential Function
IF	3	If [Arg0] then return [Then] branch ([Arg1]), otherwise return [Else] branch ([Arg2])
$\leq$, $\geq$	2	Less or equal, greater or equal
&&, ||	2	Logical AND, logical OR

Terminals		
Name	**Parameters**	**Description**
var	x, c	Value of attribute x multiplied with coefficient c
const	d	A constant double value d

15.1.2 Solution Representation Using Hybrid Tree Structures

The selection of the functions library is an important part of any GP modeling process because this library should be able to represent a wide range of systems; Table 15.1 gives an overview of the function set as well as the terminal nodes used for the classification experiments documented here. As we can see in Table 15.1, mathematical functions and terminal nodes are used as well as Boolean operators for building complex arithmetic expressions. Thus, the concept of decision trees is included in this approach together with the standard structure identification concept that tries to evolve nonlinear mathematical expressions. An example showing the structure tree representation of a combined formula including arithmetic as well as logical functions is displayed in Figure 15.1.

15.1.3 Evaluation of Classification Models

There are several possible functions that can serve as fitness functions within the GP process. For example, the ratio of misclassifications (using optimal thresholds)

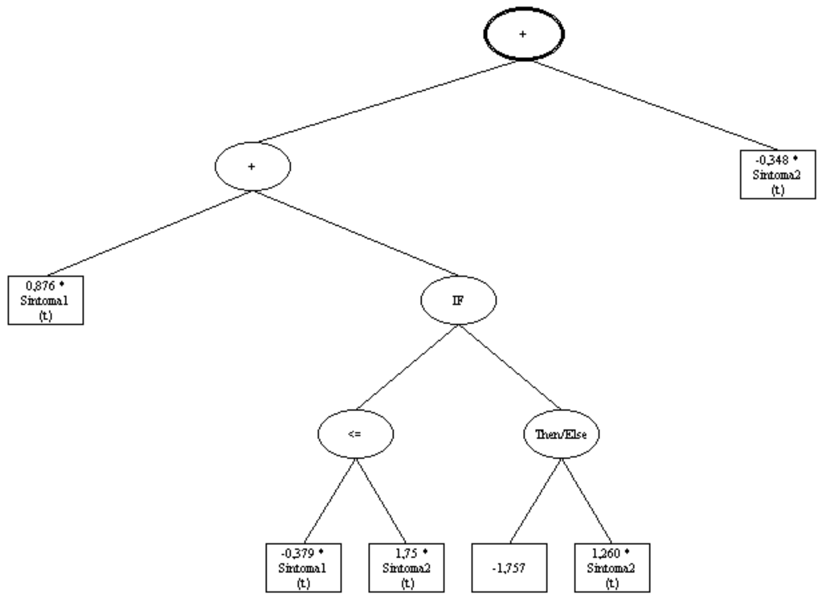


Figure 15.1: An exemplary hybrid structure tree.

or the area under the corresponding ROC curves ([ZC93], [Bra97]) could be used. Another function frequently used for quantifying the quality of models is the R^2 function that takes into account the sum of squared errors as well as the sum of squared target values; an alternative, the so-called *adjusted* R^2 function, is also utilized in many applications.

We have decided to use a *variant of the squared errors function* for estimating the quality of a classification model. There is one major difference of this modified mean squared errors function to the standard implementation of this function: The errors of predicted values that are lower than the lowest class value or greater than the greatest class value do not have a totally quadratic, but partially only linear contribution to the fitness value. To be a bit more precise: Given N samples with original classifications o_i divided into n classes $c_1, ..., c_n$ (with c_1 being the lowest and c_n the greatest class value), the fitness value F of a classification model producing the estimated classification values e_i is evaluated as follows:

$$\begin{aligned}
\forall(i \in [1, N]) : & \\
(e_i < c_1) \Rightarrow f_i &= (o_i - c_1)^2 + \mid c_1 - e_i \mid, \\
(c_1 \leq e_i \leq c_n) \Rightarrow f_i &= (e_i - o_i)^2, \\
(e_i > c_n) \Rightarrow f_i &= (o_i - c_n)^2 + \mid c_n - e_i \mid
\end{aligned} \tag{15.1}$$

$$F = \frac{1}{N}\sum_{i=1}^{N} f_i \tag{15.2}$$

The reason for this is that values that are greater than the greatest class value or below the lowest value are anyway classified as belonging to the class having the greatest or the lowest class number, respectively; using a standard implementation of the squared error function would punish a formula producing such values more than necessary.

15.1.4 Finding Appropriate Thresholds: Dynamic Range Selection

Of course, a mathematical expression alone does not yet define a classification model; thresholds are used for dividing the output into multiple ranges, each corresponding to exactly one class. These regions are defined before starting the training algorithm in static range selection (SRS, see for example [PLC05] for explanations), which brings along the difficulty of determining the appropriate range boundaries a priori. In the GP based classification framework discussed in this paper we have therefore used dynamic range selection (DRS) which attempts to overcome this problem by evolving the range thresholds along with the classification models: Thresholds are chosen so that the sum of class-wise ratios of misclassifications for all given classes is minimized (on the training data, of course).

In detail, let us consider the following: Given N (training) samples with original classifications o_i divided into n classes $c_1, \ldots, c_n$ (with c_1 being the lowest and c_n the greatest class value), models produced by GP can be in general used for calculating estimated values e_i for all N samples. Assuming thresholds $T = t_1, \ldots, t_{n-1}$ (with $c_j < t_j < c_{j+1}$ for $j \in [1; n-1]$), each sample k is classified as ec_k:

$$\begin{aligned} e_k < t_1 \quad &\Rightarrow \quad ec_k(T) = c_1 && (15.3)\\ t_j < e_k < t_{j+1} \quad &\Rightarrow \quad ec_k(T) = c_{j+1} && (15.4)\\ e_k > t_{n-1} \quad &\Rightarrow \quad ec_k(T) = c_n && (15.5) \end{aligned}$$

Thus, assuming a set of thresholds T_m, for each class c_k we get the ratio of correctly classified samples cr_k as

$$\begin{aligned} total_k(T_m) &= |\{a : (\forall (x \in a) : o_x = ec_k(T_m))\}| && (15.6)\\ correct_k(T_m) &= |\{b : (\forall (x \in b) : o_x = ec_k(T_m) \wedge e_x = c_k)\}| && (15.7)\\ cr_k(T_m) &= \frac{correct_k(T_m)}{total_k(T_m)}. && (15.8) \end{aligned}$$

The sum of ratios of correctly classified samples is – dependent on the set of thresholds T_m – calculated as

$$cr(T_m) = \sum_{i=1}^{n} cr_i(T_m). \tag{15.9}$$

So, finally we can define the set of thresholds applied as that set T_{opt} so that each other set of thresholds leads to lower sums of classification accuracies:

$$T_d \neq T_{opt} \Rightarrow cr(T_d) \leq cr(T_{opt}) \tag{15.10}$$

These thresholds, that are optimal for the training samples, are fixed and also applied on the test samples.

Please note that this sum of class-wise classification accuracies is not equal to the total ratio of correctly classified samples which is used later on in Sections 15.1.5 and 15.1.8; the total classification accuracy for a set of thresholds $acc(T_m)$ (assuming original and estimated values $\mathbf{o}$ and $\mathbf{e}$) is defined as

$$z(T_m) = |\{a|(\forall(x \in a) : o_x = ec_x(T_m))\}| \tag{15.11}$$

$$acc(T_m) = \frac{z}{N}. \tag{15.12}$$

15.1.5 First Results and Optimal Parameter Settings

As first reported in detail in [WAW07a], during our thorough test series we have identified the following GP-relevant parameter settings as the best ones for solving classification problem instances:

- **GP-algorithm:** Enhanced GP using strict offspring selection.
- **Mutation rate:** 10% – 15%.
- **Population size:** 500 – 2,000.
- **Selection operators:** Whereas standard GA implementations use only one selection operator, the SASEGASA requires two, namely the so-called female selection operator as well as the male selection operator. Similar to our experience gained during the tests on the identification of mechatronical systems, it seems to be the best to choose the roulette-wheel selection in combination with the random selection operator. The reason for this is that apparently merging the genetic information of rather good individuals (models, formulas) with randomly chosen ones is the best strategy when using the SASEGASA for solving identification problems.

- **Success ratio and selection pressure:** As for instance described in [AW04], there are some additional parameters of the SASEGASA regarding the selection of those individuals that are accepted to be a part of the next generation's population. These are the success ratio and the maximal selection pressure that steer the algorithm's behavior regarding offspring selection. For model structure identification tasks in general and especially in case of dealing with classification problems, the following parameter settings seem to be the best ones:

  – Success ratio = 1.0, and

  – Maximum selection pressure = 100 – 500 (this value has to be defined before starting a identification process depending on other settings of the genetic algorithm used and the problem instance which is to be solved).

  As has already been explained in further detail in previous chapters, these settings have the effect that in each generation only offspring survive that are really better than their parent individuals (since the success ratio is set to 1.0, only better children are inserted into the next generation's population). This is why the selection pressure becomes very high as the algorithm is executed, and therefore the maximum selection pressure has to be set to a rather high value (as, e.g., 100 or 500) to avoid premature termination.

- **Crossover operators:** We have implemented and tested three different single-point crossover procedures for GP-based model structure identification: One that exchanges rather big subtrees, one that is designed to exchange rather small structural parts (e.g., only one or two nodes) and one that replaces randomly chosen parts of the respective structure trees. Moreover, for each crossover operator we have also implemented an extended version that additionally randomly mutates all terminal nodes (i.e., manipulates the parameters of the represented formula). The following 6 structure identification crossover operators are available: *StandardSPHigh*, *StandardSPMedium*, *StandardSPLow*, *ExtendedSPHigh*, *ExtendedSPMedium*, and *ExtendedSPLow*.
  Since arbitrarily many crossover operators can be selected when applying the SASEGASA[2], the task was not to find out which operator can be used to produce the best results but rather which subset of operators is to be chosen. According to what we experienced, the following set of crossover operators

[2]Using more than one crossover operator within the SASEGASA does not mean using a combination of several operators for creating one new solution, but rather in the following way: Every time a new child is to be produced using two parent individuals, one of the given crossover operators is chosen randomly; the chance of being applied is equal for each operator.

should be applied: All three standard operators (*StandardSPHigh*, *StandardSPMedium* and *StandardSPLow*) plus one of the extended ones, for instance *ExtendedSPLow*.

- **Mutation operators:** The basic mutation operator for GP structure identification we have implemented and tested, *GAStandard*, works as already described: A function symbol could become another function symbol or be deleted, the value of a constant node or the index of a variable could be modified. Furthermore, we have also implemented an extended version (*GAExtended*) that additionally randomly mutates all terminal nodes (in analogy to the extended crossover operators).
  As the latest test series have shown, the choice of the crossover operators influences the decision which mutation operator to apply to the SASEGASA: If one of the extended crossover operators is selected, it seems to be the best to choose the standard mutation operator. But if only standard crossover methods are selected, picking the extended mutation method yields the best results.

Selected experimental results of the standard GP implementation and the SASEGASA algorithm for the *Thyroid* data set using various parameter settings are presented in Table 15.2. For each parameter settings version the 10-fold cross validation test runs were executed, the resulting average results are listed. In all cases, the population size was 1000; furthermore, the following parameter settings were used:

- **(1)**: crossover: *ExtendedSPMedium*; mutation: *GAStandard*; selection: *roulette*.
- **(2)**: crossover: *StandardSPMedium*; mutation: *GAExtended*; selection: *roulette*.
- **(3)**: crossover: *all 6 available operators*; mutation: *GAExtended*; selection: *random* and *roulette* (maximum selection pressure: 500).
- **(4)**: crossover: *all 6 available operators*; mutation: *GAStandard*; selection: *Random* and *roulette* (maximum selection pressure: 500).
- **(5)**: crossover: *all 3 standard operators plus ExtendedSPLow*; mutation: *GAStandard*; selection: *roulette* and *roulette* (maximum selection pressure: 500).
- **(6)**: crossover: *all 3 standard operators plus ExtendedSPLow*; mutation: *GAStandard*; selection: *random* and *roulette* (maximum selection pressure: 500).

Using standard GP implementation		
Parameter	*Correct classifications*	
settings	*Evaluation*	*Prognosis*
(1)	92.80%	92.13%
(2)	93.91%	93.25%
Using the SASEGASA		
Parameter	*Correct classifications*	
settings	*Evaluation*	*Prognosis*
(3)	97.15%	96.34%
(4)	98.21%	98.07%
(5)	97.70%	97.25%
(6)	**98.93%**	**98.53%**

Table 15.2: Experimental results for the *Thyroid* data set.

As an example, the model produced for cross validation partition 3 using the parameter settings combination (6) is shown in Figure 15.5.

Parameter	*Optimal Value*
GP algorithm	SASEGASA (GP with offspring selection)
Mutation rate	10% – 15%
Population size	1,000
Selection operators	Random, roulette
Maximum selection pressure	100 – 1,000
Crossover Operators	StandardSPLow, StandardSPMedium, StandardSPHigh, ExtendedSPLow
Mutation operator	GAStandard
Ratio of weighting the evaluation contributions SumOfSquaredErrors : separability : class ranges	4 : 1: 1

Table 15.3: Summary of the best GP parameter settings for solving classification problems.

These insights have been used also in the more extensive test series documented later on in this chapter.

15.1.6 Graphical Classifier Analysis

Graphical analysis can often help analyzing results achieved to any kind of problem; this is of course also the case in machine learning and in data-based classification.

The most common and also simplest way how to illustrate classification results is to plot the target values and the estimated values into one chart; Figure 15.2 shows a graphical representation of the best result obtained for the *Thyroid* data set, cross-validation set 9.

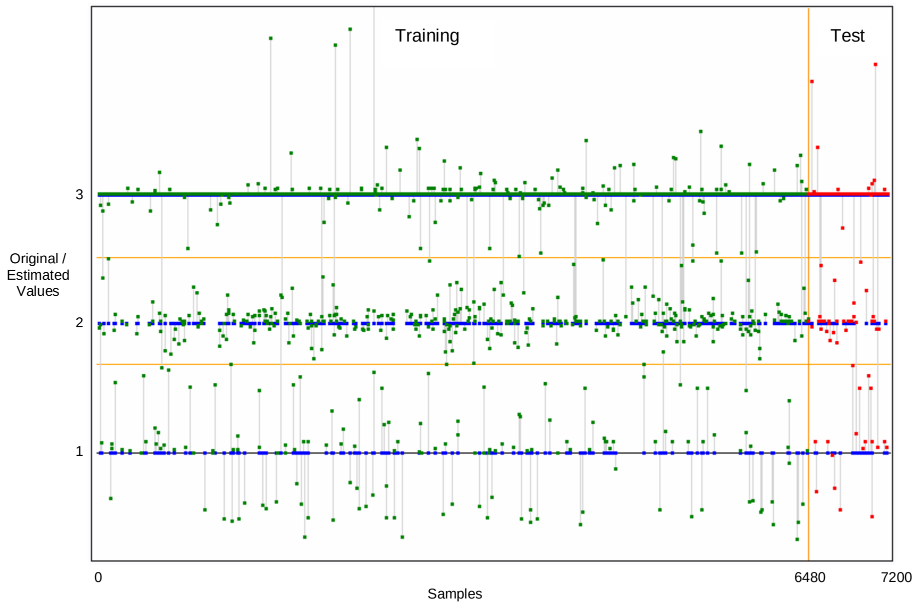


Figure 15.2: Graphical representation of the best result we obtained for the *Thyroid* data set, CV-partition 9: Comparison of original and estimated class values.

In Figure 15.3 we show 4 ROC charts examples that were generated for the classes '0' and '2' of the *Thyroid* data set, 10-fold cross validation set number 9:

- (a) ROC curve for an unsuitable classifier for class '2', evaluated on training data;
- (b) ROC curve for the best identified classifier for class '0', evaluated on training data;
- (c) ROC curve for the best identified classifier for class '0', evaluated on test data;

- (d) ROC curve for the best identified classifier for class '2', evaluated on test data.

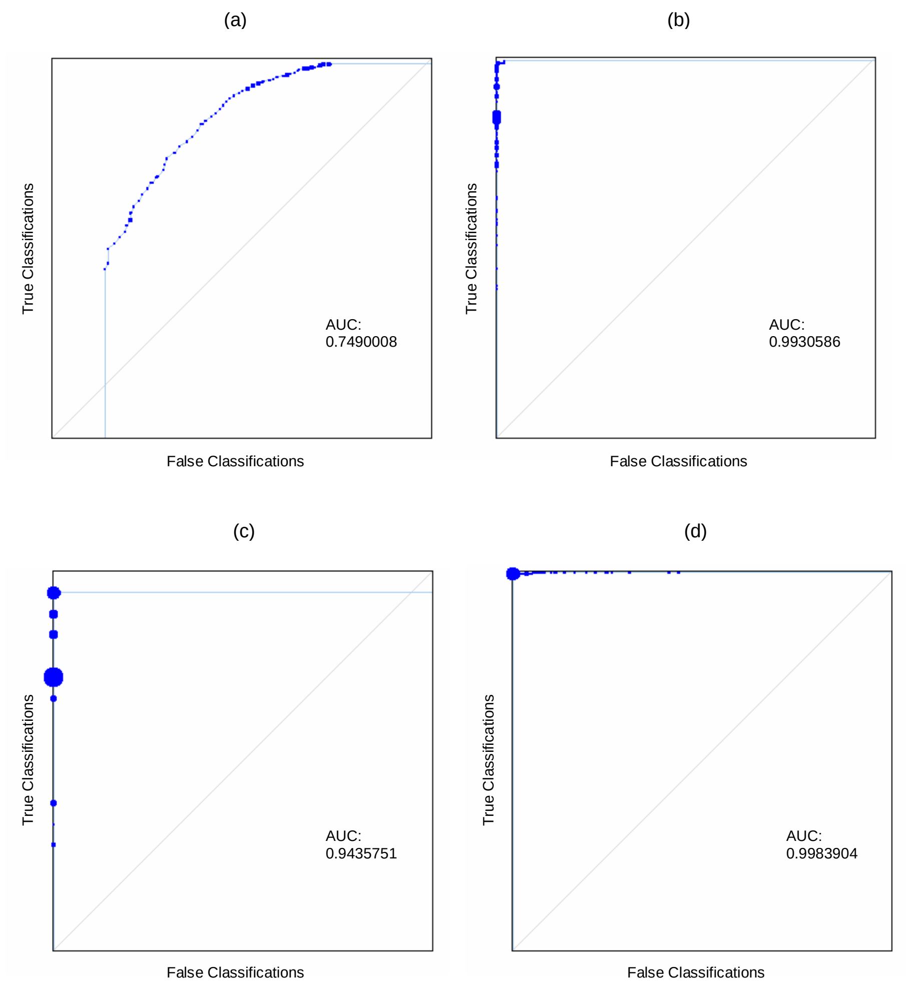


Figure 15.3: ROC curves and their area under the curve (AUC) values for classification models generated for *Thyroid* data, CV-set 9.

In Figure 15.4 finally we show 4 MROC charts examples that were generated for the intermediate classes '1' of the *Thyroid* data set, again on the basis of 10-fold CV-set number 9:

- (a) MROC curve for an unsuitable classifier for class '1', evaluated on training data;
- (b) MROC curve for an unsuitable classifier for class '1', evaluated on test data;

- (c) MROC curve for the best identified classifier for class '1', evaluated on training data;
- (d) MROC curve for the best identified classifier for class '1', evaluated on test data.

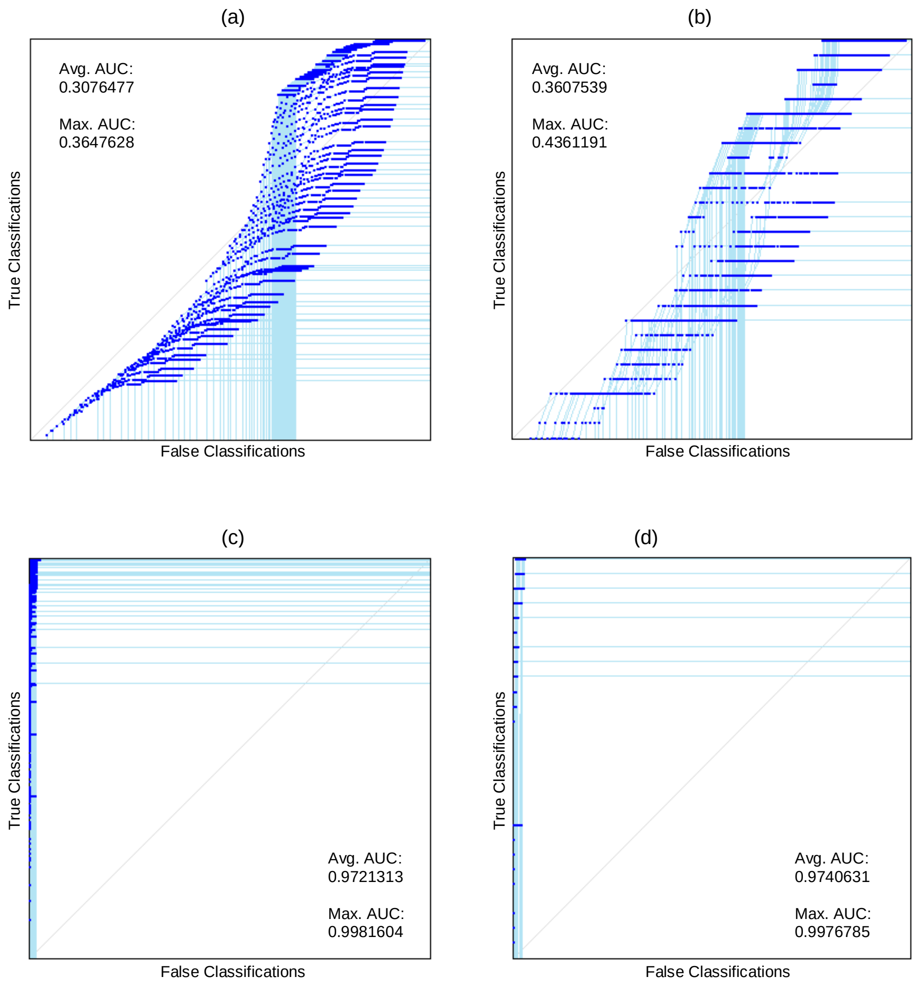


Figure 15.4: MROC charts and their maximum and average area under the curve (AUC) values for classification models generated for *Thyroid* data, CV-set 9.

Figure 15.6 finally shows a collection of 10 example models (exactly one for each partition of the 10-fold cross-validation) for the *Thyroid* data set, produced by GP. Optimal thresholds as well as resulting classification results are also given as respective confusion matrices. Please note: All variables were linearly scaled to the interval $[0; 100]$, the threshold values are therefore also values between 0 and 100.

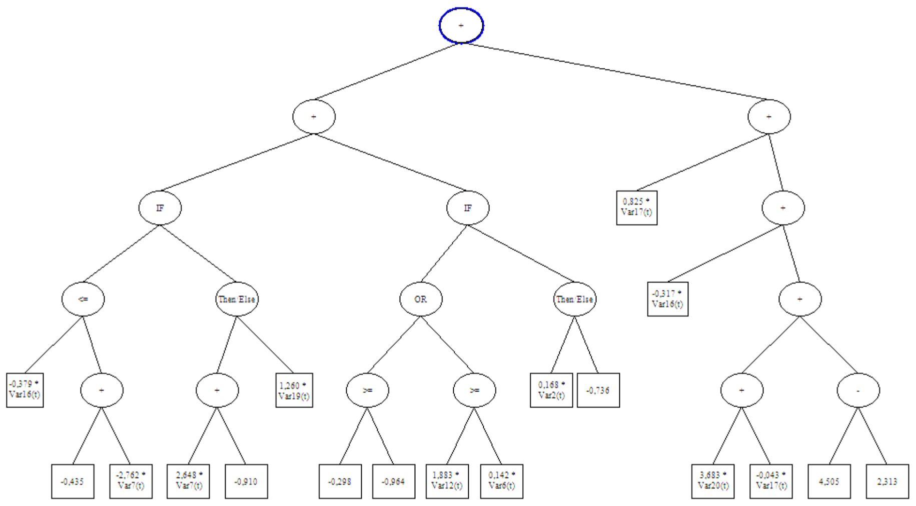


Figure 15.5: Graphical representation of a classification model (formula), produced for 10-fold cross validation partition 3 of the *Thyroid* data set.

15.1.7 Classification Methods Applied in Detailed Test Series

For comparing GP based classification with other machine learning methods, the following techniques for training classifiers were examined: Genetic programming (enhanced approach using extended parents and offspring selection), linear modeling, neural networks, the k-nearest-neighbor method, and support vector machines.

15.1.7.1 GP-Based Training of Classifiers

We have used the following parameter settings for our GP test series:

- Single population approach; population size: 500 – 1000
- Mutation rate: 10%
- Maximum formula tree height: 8
- Parents selection: Gender specific, random and roulette
- Offspring selection: Strict offspring selection (success ratio as well as comparison factor set to 1.0)

- 1-elitism
- Termination criteria:
  - Maximum number of generations: 1000; not reached, all executions were terminated via the
  - Maximum selection pressure: 100
- Function set: All functions as described in Table 15.1.
- Fitness functions:
  - In order to keep the computational effort low, the mean squared errors function with early abortion was used as fitness function for the GP training process.
  - The eventual selection of models is done by choosing those models that perform best on validation data (or, if no validation samples are specified, then the models' performance on training data is considered). For this selection we have used the classification specific evaluation function described in Section 8.2: The mean squared error is considered as well as class ranges, thresholds qualities and AUC values, all other possible contributions have been neglected in the test series reported and discussed here. Thus, $c_1 = 4.0$, $c_k = 1.0$ for $k \in \{6, 7, 8\}$, and $c_k = 0.0$ for $k \in \{2, 3, 4, 5\}$.

In addition to splitting the given data into training and test data, extended GP based training is implemented in such a way that a part of the given training data is not used for training models and serves as validation set; in the end, when it comes to returning classifiers, the algorithm returns those models that perform best on validation data. This approach has been chosen because it is assumed to help to cope with overfitting; it is also applied in other GP based machine learning algorithms as for example described in [BL04]. In fact, this was also done in our standard GP tests for the *Melanoma* data set.

15.1.7.2 Linear Modeling

Given a data collection including m input features storing the information about N samples, a linear model is defined by the vector of coefficients $\theta_{1...m}$. For calculating the vector of modeled values e using the given input values matrix $u_{1\ldots m}$, these input values are multiplied with the corresponding coefficients and added:

$$e = u_{1\ldots m} * \theta \tag{15.13}$$

The coefficients vector can be computed by simply applying matrix division. For conducting the test series documented here we have used the matrix division function provided by MATLAB©:

```
theta = InputValues \ TargetValues;
```

If a constant additive factor is to be included into the model (i.e., the coefficients vector), this command has to be extended:

```
r = size(InputValues,1);
theta = [InputValues ones(r,1)] \ TargetValues;
```

Theoretical background of this approach can be found in [Lju99].

15.1.7.3 Neural Networks

For training artificial neural network (ANN) models, three-layer feed-forward neural networks with one output neuron were created using the backpropagation as well as the Levenberg-Marquardt training method. Theoretical background and details can be found in [Nel01] (Chapter 11, "Neural Networks"), [Mar63], [Lev44] or [GMW82].

The following two approaches have been applied for training neural networks:

- On the one hand we have trained networks with 5 neurons in the hidden layer (referred to as "NN1" in the test series documentation in Section 15.1.8) as well as networks with 10 hidden neurons (referred to as "NN2" in the test series documentation); the number of iterations of the training process was set to 100 (in the first variant, "NN1") and 300 (in the second variant, "NN2"). In the context of analyzing the benchmark problems used here, higher numbers of nodes or iterations are likely to lead to overfitting (i.e., a better fit on the training data, but worse test results).
  The ANN training framework used to collect the results reported in this paper is the `NNSYSID20` package, a neural network toolbox implementing the Levenberg-Marquardt training method for MATLAB©; it has been implemented by Magnus Nørgaard at the Technical University of Denmark [Nør00].

- On the other hand, the multilayer perceptron training algorithm available in WEKA [WF05] has also been used for training classifiers. In this case the number of hidden nodes was set to $(a+c)/2$, where a is the number of attributes (features) and c the number of classes. The number of iterations was not pre-defined, but 10% of the training data were designated to be used as validation data; in order to combat the danger of overfitting, the training algorithm was terminated as soon as the error on validation data is constantly getting worse in 20 iterations consecutively. This training method, which applies backpropagation learning, is in the following referred to as the "NN3" method.

15.1.7.4 kNN Classification

Unlike other data based modeling methods based on linear models, neural networks or GP, k-nearest-neighbor classification works without creating any explicit models. During the training phase, the data are simply collected; when it comes to classifying a new, unknown sample x_{new}, the sample-wise distance between x_{new} and all other training samples x_{train} is calculated and the classification is done on the basis of those k training samples (x_{NN}) showing the smallest distances from x_{new}.

The distance between two samples is calculated as follows: First, all features are normalized by subtracting the respective mean values and dividing the remaining samples by the respective variables' standard deviation. Given a data matrix x including m features storing the information about N samples, the normalized values x_{norm} are calculated as

$$\forall(i \in [1,m])\forall(j \in [1,N]) : x_{norm}(i,j) = \frac{x(i,j) - \frac{1}{N}\sum_{k=1}^{N} x(i,k)}{\sigma(x(i,1\dots N))} \tag{15.14}$$

where the standard deviation σ of a given variable x storing N values is calculated as

$$\sigma(x) = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2} \tag{15.15}$$

with $\bar{x}$ denoting the mean value of x.
Then, on the basis of the normalized data, the distance between two samples a and b, $d(a,b)$, is calculated as the mean squared variable-wise distance:

$$d(a,b) = \frac{1}{n}\sum_{i=1}^{n}(a_{norm}(i) - b_{norm}(i))^2 \tag{15.16}$$

where n again is the number of features stored for each sample.

In the context of classification, the numbers of instances (of the k nearest neighbors) are counted for each given class and the algorithm automatically predicts that class that is represented by the highest number of instances. In the test series documented in this paper we have applied weighting to kNN classification: The distance between x_{new} and any sample x_z is relevant for the classification statement, the weight of "nearer" samples is higher than that of samples that are "further" away from x_{new}.

There is a lot of literature that can be found for kNN classification; very good explanations and compact overviews of kNN classification (including several possible variants and applications) are for example given in [DHS00] and [RN03].

15.1.7.5 Support Vector Machines

Support vector machines (SVMs) are a widely used approach in machine learning based on statistical learning theory [Vap98]; an example of the application of SVMs in the medical domain has been reported in [MIB$^+$00], e.g.

The most important aspect of SVMs is that it is possible to give bounds on the generalization error of the models produced, and to select the respectively best model from a set of models following the principle of structural risk minimization [Vap98]. SVM are designed to calculate hyperplanes that separate the data from each other and maximize the margin between sets of data points. While the basic training algorithm is only able to construct linear separators, so-called kernel functions can be used to calculate scalar products in higher-dimensional spaces; if the kernel functions used are non-linear, then the separating boundaries will be non-linear, too.

In this work we have used the SVM implementation described in [Pla99] and [KSBK01]; we have used the implementation of this algorithm which is available for the WEKA machine learning framework [WF05]. Polynomial kernels have been used as well as Gaussian radial basis function kernels with the γ parameter (defining the inverse variance) set to 0.01 and the complexity parameter c set to 10,000.

15.1.8 Detailed Test Series Results

Since the *Wisconsin* and the *Thyroid* data sets are publicly available, the results produced by GP are compared to those that have been published previously for

various machine learning methods; the *Melanoma* is not openly available, therefore we have used all machine learning approaches mentioned for training classifiers for this data set.
All three data sets were investigated via 10-fold cross-validation (CV). For each data collection, each of the resulting 10 pairs of training and test data partitions has been used in 5 independent GP test runs; for the *Melanoma* data set, all machine learning algorithms mentioned previously have also been applied to all pairs of training and test data, the stochastic algorithms again applied 5 times independently.

The results summarized in this section have been partially published in [WAW06b], [WAW06e] and [WAW07a].

15.1.8.1 Results for the *Wisconsin* Data Set

Table 15.4 summarizes the results for the 10-fold cross validation produced by GP with offspring selection as described in Section 15.1.7.1. These figures boil down to the fact that extended GP has in this case been able to produce classifiers that on average correctly classify 97.91% of training samples and 97.53% of test samples.

Table 15.4: Summary of training and test results for the *Wisconsin* data set: Correct classification rates (average values and standard deviation values) for 10-fold CV partitions, produced by GP with offspring selection.

Partition	**Training**		**Test**	
	Avg.	*Std.Dev.*	*Avg.*	*Std.Dev.*
0	97.69%	0.27	97.06%	1.04
1	97.69%	0.85	97.65%	2.23
2	98.40%	0.72	97.94%	1.32
3	98.37%	0.56	98.24%	1.23
4	97.52%	0.78	97.06%	2.08
5	97.95%	0.77	97.94%	1.32
6	98.05%	0.43	97.05%	1.47
7	98.05%	0.47	97.65%	1.68
8	97.75%	0.62	97.65%	1.32
9	97.62%	0.74	97.06%	1.47
Avg.	**97.91%**	**0.62**	**97.53%**	**1.51**

In order to compare the quality of these results to those reported in the literature, Table 15.5 summarizes test accuracies that have been obtained using 10-fold cross

validation. For each method listed we give the references to the respective articles in which these results have been reported[3]. Obviously the results summarized in Table 15.4 have to be considered surprisingly good as they outperform all other algorithms reported in the literature listed here.
In [PLC05], for example, recent results for several classification benchmark problems are documented; the *Wisconsin* data set was there analyzed using standard GP as well as three other GP based classification variants (POPE-GP, DecMO-GP and DecMOP-GP), and the respective results are also listed in Table 15.5.

Of course, for the sake of honesty we have to admit that the effort of GP to produce these classifiers is higher than the runtime or memory consumed by most other machine learning algorithms; in our GP tests using the *Wisconsin* data set and populations with 500 individuals the average number of generations executed was 51.6 and the average number of solutions evaluated ~1,296,742.

Table 15.5: Comparison of machine learning methods: Average test accuracy of classifiers for the *Wisconsin* data set.

Algorithm	**Test Accuracy**
GP with OS	*97.53%*
Probit [WHMS03]	97.20%
RLP [BU95]	97.07%
SVM [WHMS03]	96.70%
C4.5 (decision tree) [HSC96]	96.0%
ANN [TG97]	95.61%
DecMOP-GP [PLC05]	95.60%
DecMO-GP [PLC05]	95.19%
POPE-GP [PLC05]	95.08%
StandardGP [PLC05]	93.82%

15.1.8.2 Results for the *Melanoma* Data Set

For the *Melanoma* data set no results are available in the literature, therefore we have tested all machine learning algorithms mentioned previously for getting an objective evaluation of our GP methods.

First, in Table 15.6 we summarize original vs. estimated classifications obtained by applying the classifiers produced by GP with offspring selection; in total, 97.17%

[3]An even more detailed listing of test results for this data set can be found in [JHC04].

of the training and 95.42% of the test samples are classified correctly (with standard deviations 0.87 and 2.13, respectively). These GP tests using the *Melanoma* data set were done with populations containing 1,000 individuals; the average number of generations executed was 54.4 and the average number of solutions evaluated ∼2,372,629.

Table 15.6: Confusion matrices for average classification results produced by GP with OS for the *Melanoma* data set.

Training		*Original Classification*	
		[0] (Benign)	*[1] (Malign)*
Estimated	*[0]*	1,043.21 (88.41%)	9.09 (0.77%)
Classification	*[1]*	24.28 (2.06%)	103.42 (8.76%)
Test		*Original Classification*	
		[0] (Benign)	*[1] (Malign)*
Estimated	*[0]*	115.18 (87.92%)	2.67 (2.04%)
Classification	*[1]*	3.33 (2.54%)	9.82 (7.50%)

Test results obtained using other machine learning algorithms are collected in Table 15.7. Support vector machine based training was done with radial as well as with polynomial kernel functions, furthermore we used γ values 0.001 and 0.01. In standard GP (SGP) tests we used tournament parents selection ($k = 3$), 8% mutation, single point crossover and the same structural limitations as in GP with OS; in order to get a fair comparison, the population size was set to 1,000 and the number of generations to 2,500 yielding 2,500,000 evaluations per test run.

As we can see in Table 15.7, our GP implementation performs approximately as well as the support vector machines and neural nets applying those settings that are optimal in this test case: GP with OS was able to classify 95.42% of the test cases correctly, SVMs correctly classified 94.89% – 95.47% and neural nets (with validation set based stopping) 95.27% of the test cases evaluated. Standard GP as well as kNN, linear regression and standard ANNs clearly perform worse.
Even though it is nice to see that the average accuracy recorded for models produced by GP with OS is quite fine, the relatively standard deviation of this method's performance (2.13, compared to 0.41 recorded for optimal SVMs) has to be seen as a negative aspect of these results.

Table 15.7: Comparison of machine learning methods: Average test accuracy of classifiers for the *Melanoma* data set.

Algorithm	**Test Accuracy**	
	Avg.	*Std.Dev.*
SVM (radial, $\gamma = 0.01$)	95.47%	0.41
GP with OS	*95.42%*	*2.13*
SVM (polynomial, $\gamma = 0.01$)	95.40%	0.56
SVM (radial, $\gamma = 0.001$)	95.27%	0.74
NN3	95.27%	1.91
SVM (polynomial, $\gamma = 0.001$	94.89%	0.83
NN1	94.35%	2.39
kNN ($k = 3$)	93.59%	1.03
SGP	93.52%	3.72
NN2	92.90%	2.59
kNN ($k = 5$)	92.85%	0.94
Lin	92.45%	2.90

15.1.8.3 Results for the *Thyroid* Data Set

Finally, the results achieved for the *Thyroid* data set are to be reported here. Table 15.8 summarizes the results for the 10-fold cross validation produced by GP with offspring selection as described in Section 15.1.7.1. For each class we characterize the classification accuracy on training and test data, giving average as well as standard deviation values for each partition. These figures boil down to the fact that extended GP has in this case been able to produce classifiers that on average correctly classify 99,10% of training samples and 98,76% of test samples, the total standard deviation values being 0.73 and 0.92, respectively.

In order to compare the quality of these results to those reported in the literature, Table 15.9 summarizes a selection of test accuracies that have been obtained using 10-fold cross validation; again, for each method listed we give the references to the respective articles in which these results have been reported. Obviously, the results summarized in Table 15.8 have to be considered quite fine, but not perfect as they are outperformed by results reported in [WK90] and [DAG01].

GP has also been repeatedly applied for solving the *Thyroid* problem, some of the results published are the following ones:
In [LH06] (Table 8), results produced by a pareto-coevolutionary GP classifier sys-

Table 15.8: Summary of training and test results for the *Thyroid* data set: Correct classification rates (average values and standard deviation values) for 10-fold CV partitions, produced by GP with offspring selection.

Partition		**Training**			**Test**		
		Class 1	*Class 2*	*Class 3*	*Class 1*	*Class 2*	*Class 3*
0	*avg.*	94.67%	97.64%	99.63%	90.00%	95.68%	99.19%
	std.dev.	1.70	2.65	0.52	7.13	4.10	0.64
1	*avg.*	94.93%	98.67%	99.01%	88.75%	96.76%	99.43%
	std.dev.	3.58	4.23	0.46	5.23	5.86	0.44
2	*avg.*	96.67%	98.49%	99.49%	91.25%	96.22%	97.90%
	std.dev.	1.89	2.00	0.55	3.42	6.51	2.02
3	*avg.*	96.00%	98.19%	99.15%	90.00%	95.68%	99.46%
	std.dev.	2.87	1.56	0.42	5.59	4.10	0.31
4	*avg.*	95.33%	97.04%	99.19%	88.75%	96.22%	99.61%
	std.dev.	2.45	5.38	0.35	11.18	3.63	0.27
5	*avg.*	95.07%	96.62%	99.22%	95.00%	94.59%	99.37%
	std.dev.	1.92	5.29	0.40	5.23	4.27	0.29
6	*avg.*	93.47%	97.76%	99.16%	87.50%	94.59%	98.56%
	std.dev.	2.18	7.64	0.49	7.65	4.27	0.91
7	*avg.*	98.80%	98.97%	99.16%	87.50%	92.97%	99.40%
	std.dev.	2.18	5.92	0.49	7.65	4.52	0.30
8	*avg.*	94.40%	98.01%	99.23%	96.25%	94.05%	99.34%
	std.dev.	5.11	4.99	0.64	5.23	3.52	0.57
9	*avg.*	97.73%	96.62%	99.31%	91.25%	92.43%	99.55%
	std.dev.	2.69	2.65	0.52	3.42	3.52	0.15
Avg.	*avg.*	95.71%	97.80%	99.26%	90.63%	94.92%	99.18%
	std.dev.	2.66	4.23	0.48	6.17	4.43	0.59

tem for the *Thyroid* problem are reported, and here in Table 15.9 these results are stated as the "PGPC" results; in fact, these results are not the mean accuracy values but rather the median value, which is why these results are not totally comparable to other results stated here. Loveard and Ciesielski [LC01] reported that classifiers for the *Thyroid* problem could be produced using GP with test accuracies ranging from 94.9% to 98.2% (depending on the range selection strategy used).
According to Banzhaf and Lasarczyk [BL04], GP-evolved programs consisting of register machine instructions turned out to eventually misclassify on average 2.29% of the given test samples, and that optimal classifiers are able to correctly classify

98.64% of the test data.
Furthermore, Gathercole and Ross [GR94] report classification errors between 1.6% and 0.73% as best result using tree-based GP, and that a classification error of 1.52% for neural networks is reported in [SJW92]. In fact, Gathercole and Ross reformulated the *Thyroid* problem to classifying cases as "class 3" or "not class 3"; as is stated in [GR94], it turned out to be relatively straight forward for their GP implementation (DSS-GP) to produce function tree expressions which could distinguish between classes "1" and "2" completely correctly on both the training and test sets. "To be fair, in splitting up the problem into two phases (class *3* or not, then class *1* or *2*) the GP has been presented with an easier problem [...]. This could be taken in different ways: Splitting up the problem is mildly cheating, or demonstrating the flexibility of the GP approach." (Taken from [GR94].)

Table 15.9: Comparison of machine learning methods: Average test accuracy of classifiers for the *Thyroid* data set.

Algorithm	**Accuracy**	
	Training	**Test**
CART [WK90]	99.80%	99.36%
PVM [WK90]	99.80%	99.33%
Logical Rules [DAG01]	–	99.30%
GP [GR94]	–	98.4% – 99.27%
GP with OS	*99.10%*	*98.76%*
GP [BL04]	–	97.71% – 98.64%
GP [LC01]	–	94.9% – 98.2%
BP + local adapt. rates [SJW93]	99.6%	98.5%
ANN [SJW92]	–	98.48%
BP + genetic opt. [SJW93]	99.4%	98.4%
Quickprop [SJW93]	99.6%	98.3%
RPROP [SJW93]	99.6%	98.0%
PGPC [LH06]	–	97.44%

GP with strict offspring selection was here applied with populations of 1000 individuals; on average, the number of generations executed in our GP tests for the *Thyroid* test studies was 73.9, and on average 2,463,635.1 models were evaluated in each GP test run.

15.1.9 Conclusion

In this section we have presented an enhanced genetic programming method that was successfully used for investigating machine learning problems in the context of medical classification. The approach works with hybrid formula structures combining logical expressions (as used for example in decision trees) and classical mathematical functions; the enhanced selection scheme originally successfully applied for solving combinatorial optimization problems using genetic algorithms was also applied yielding high quality results.

We have intensively investigated GP in the context of learning classifiers for three medical data collections, namely the *Wisconsin* and the *Thyroid* data sets taken from the UCI machine learning repository and the *Melanoma* data set, a collection that represents medical measurements which were recorded while investigating patients potentially suffering from skin cancer. The results presented in this section are indeed satisfying and make the authors believe that an application in a real-world framework in the context of medical data analysis using the techniques presented here is recommended. As documented in the test results summary, our GP based classification approach is able to produce results that are – in terms of classification accuracy – at least comparable to or even better than the classifiers produced by classical machine learning algorithms frequently used for solving classification problems, namely linear regression, neural networks, neighborhood based classification or support vector machines as well as other GP implementations that have been used on the data sets investigated in our test studies.

10-fold CV set 0:

```
class(t) = IF(>=(Log(Log(-(Log([2.643*Var16(t)])|[1.000*Var12(t)])))|
IF(>=(-(-([1.000*Var17(t)]|[1.000*Var16(t)])|+(-0.141724|[1.000*Var18(t)]))|-
(Sig([1.000*Var12(t)])|^([1.000*Var18(t)]| [1.000*Var7(t)])))|ThenElse(-(Sin([-
2.040*Var16(t)])|
Sin([1.973*Var18(t)]))|IF(>=([1.973*Var18(t)] |49.864966)|ThenElse([2.643*Var16(t)]|[-
2.040*Var16(t)])))))|
ThenElse(+(-(+(Cos([0.427*Var18(t)])|-([1.000*Var17(t)]|1.287233))|
Sqrt(+([3.955*Var20(t)]|[11.739*Var16(t)])))|+(+(*([0.083*Var2(t)]|
[0.427*Var18(t)])|Cos([0.427*Var18(t)]))|+(*([-0.045*Var20(t)]|
[1.000*Var18(t)])|[3.955*Var20(t)])))|+(+(-2.541577|e^(/([1.973*Var18(t)]|
[0.427*Var18(t)])))|Cos(+(Sin(49.864966)|^([1.000*Var16(t)]|6.173179))))))

Thresholds: [33.45; 75.25]

TRAINING                              TEST

Orig.  -> | [0] | [1] |  [2] |        Orig.  -> | [0] | [1] | [2] |
----------|-----|-----|------|        ----------|-----|-----|-----|
Est.  [0] | 137 |   9 |    5 |        Est.  [0] |  18 |   3 |   0 |
Class [1] |   9 | 337 |   14 |        Class [1] |   0 |  15 |   0 |
      [2] |   2 |   2 | 5965 |              [2] |   0 |   2 | 682 |
----------|-----|-----|------|------- ----------|-----|-----|-----|-------
                             | 99.37%                             | 99.31%
```

10-fold CV set 1:

```
class(t) = *(+(*(-(-(Cos([1.000*Var16(t)])|Cos([-4.991*Var16(t)]))| *(/(
[1.000*Var7(t)]| 6.767502)|[-0.214*Var16(t)]))|*(*(
Sqrt([1.000*Var16(t)])|3.886609)|+(Cos([1.000*Var16(t)])|2.302652)))|-
(+(/(+([1.000*Var16(t)]|[1.000*Var2(t)])|e^([-0.214*Var16(t)]))|-(+([-
4.991*Var16(t)]|108.231865)|+(0.000000| [1.000*Var2(t)])))|-
(*(/(2.302652|1.457826)|+([1.000*Var16(t)]|3.886609)) |*(*([-4.991*Var16(t)]|[-
0.214*Var16(t)])| Sqrt([1.000*Var16(t)]))))|IF(<=(-(+(Sqrt([1.000*Var7(t)])|+
([1.000*Var11(t)]|[8.693*Var20(t)]))|+(-(0.000000|[1.006*Var20(t)])|-
(92.692883|0.000000)))|-(-(1.457826|+(0.000000| [1.000*Var4(t)]))|/(-(
[-4.991*Var16(t)]|-4.030073)|^([1.000*Var16(t)]| [8.693*Var20(t)]))))|
ThenElse(e^(-(Sin([-4.991*Var16(t)])| [1.000*Var16(t)]))|e^([-0.214*Var16(t)]))))

Thresholds: [39.225; 72.7]

TRAINING                              TEST

Orig.  -> | [0] | [1] |  [2] |        Orig.  -> | [0] | [1] | [2] |
----------|-----|-----|------|        ----------|-----|-----|-----|
Est.  [0] | 149 |   0 |   10 |        Est.  [0] |  15 |   0 |   1 |
Class [1] |   2 | 332 |   45 |        Class [1] |   0 |  36 |   1 |
      [2] |   0 |   0 | 5942 |              [2] |   0 |   0 | 667 |
----------|-----|-----|------|------- ----------|-----|-----|-----|-------
                             | 99.12%                             | 99.72%
```

10-fold CV set 2:

```
class(t) = +(+(+(IF(>=(+([2.933*Var5(t)]|[4.635*Var20(t)])|
^([1.000*Var16(t)]|32.512281))|ThenElse(104.077790|+(-
5.259905|[3.901*Var20(t)]))))|IF(>=(+([1.000*Var17(t)]|[1.000*Var17(t)])|+
([1.000*Var16(t)]|[4.635*Var20(t)]))|ThenElse(+([1.000*Var2(t)]| [4.635*Var20(t)])|-
4.370451)))|IF(>=(^(/([1.000*Var16(t)]|
[1.000*Var17(t)])|+([1.000*Var8(t)]|[0.373*Var7(t)]))|*(Sqrt([4.635*Var20(t)])|Log(
[1.000*Var16(t)])))|ThenElse([1.000*Var16(t)]|+(+(-5.259905|-4.370451)|+(-
5.259905|[0.611*Var7(t)])))))|IF(>=(+(/(+([1.000*Var10(t)]
|0.000000)|/([1.000*Var16(t)]|[1.000*Var2(t)]))|+(Sqrt([2.880*Var20(t)])|+(
[1.000*Var4(t)]|-4.370451)))|Cos(^(Log([1.000*Var16(t)])|-
(-3.260221|[1.000*Var13(t)]))))|ThenElse(*(^(Sqrt([0.079*Var20(t)])| +(
[1.000*Var14(t)]|-3.260221))|+(+([1.000*Var2(t)]|[1.000*Var17(t)])
|+([1.000*Var17(t)]|[1.000*Var16(t)])))|/(+(+([1.000*Var8(t)]|
[0.611*Var7(t)])|+([1.000*Var8(t)]|[0.373*Var7(t)]))|-
(+([2.620*Var20(t)]|[1.000*Var10(t)])|-(6.267298|[1.000*Var13(t)]))))))

Thresholds: [32.425; 76.475]

TRAINING                              TEST

Orig.  -> | [0] | [1] |  [2] |        Orig.  -> | [0] | [1] | [2] |
----------|-----|-----|------|        ----------|-----|-----|-----|
Est.  [0] | 140 |   7 |    2 |        Est.  [0] |  17 |   0 |   0 |
Class [1] |   5 | 316 |   27 |        Class [1] |   4 |  45 |   3 |
      [2] |   0 |   0 | 5983 |              [2] |   0 |   0 | 651 |
----------|-----|-----|------|------- ----------|-----|-----|-----|-------
                             | 99.37%                             | 99.03%
```

10-fold CV set 3:

```
class(t) = +(+(IF(<=([-0.37921775*Var16(t)]|+(-0.43533937|
[-2.76239437*Var7(t)]))|ThenElse( +([2.64778823*Var7(t)]|
-0.91019582)|[1.26036869*Var19(t)]))|IF(OR(>=(-0.29808521|-0.96359298)|
>=([1.88284005*Var12(t)]|[0.14167759*Var6(t)]))| ThenElse([0.16761998*Var2(t)]|-
0.73643814))) |+([0.82477057*Var17(t)]|+
([-0.31729187*Var16(t)]|+(+([3.68276008*Var20(t)]|[-0.04266986*Var17(t)])|
-(4.50479669|2.31342641)))))

Thresholds: [43.125; 89.025]

TRAINING                              TEST

Orig.  -> | [0] | [1] |  [2] |        Orig.  -> | [0] | [1] | [2] |
----------|-----|-----|------|        ----------|-----|-----|-----|
Est.  [0] | 136 |   4 |   10 |        Est.  [0] |  16 |   1 |   0 |
Class [1] |   9 | 331 |  231 |        Class [1] |   3 |  28 |  23 |
      [2] |   2 |   3 | 5754 |              [2] |   0 |   1 | 648 |
----------|-----|-----|------|------- ----------|-----|-----|-----|-------
                             | 96.00%                             | 96.11%
```

10-fold CV set 4:

```
class(t) = IF(<=(/(-(*(-0.507917|[1.000*Var8(t)])|[0.875*Var16(t)])|-
([0.253*Var2(t)]|Log([0.875*Var16(t)])))|+(*(*(+([1.000*Var4(t)]|
[0.720*Var7(t)])|Sin([0.875* Var16(t)]))|/([-2.651*Var16(t)]|-
([0.875*Var16(t)]|[1.000*Var20(t)])))|+
([-0.548*Var16(t)]|e^(e^([9.284*Var12(t)]))))))|ThenElse(-(-(-(
-(0.041530|-100.301158)|0.000000)|Sin(12.938045))|/(/([-2.651*Var16(t)]
|3.143818)|-([9.284*Var12(t)]|[1.000*Var20(t)])))|+(+(*(+([1.000*Var2(t)]
|[0.720*Var7(t)])|Sin(12.938045))|+(*([0.012*Var17(t)]|[1.000*Var17(t)])|[-
0.548*Var16(t)]))|-(-(-([5.281*Var20(t)]|[0.856*Var20(t)])|
[0.856*Var20(t)])|+(Cos([0.780*Var20(t)])|-([0.856*Var20(t)]|[1.000*Var17(t)]))))))

Thresholds: [35.55; 77.95]

TRAINING                              TEST

Orig.  -> | [0] | [1] |  [2] |        Orig.  -> | [0] | [1] | [2] |
----------|-----|-----|------|        ----------|-----|-----|-----|
Est.  [0] | 134 |   8 |    5 |        Est.  [0] |  15 |   4 |   1 |
Class [1] |  13 | 309 |   33 |        Class [1] |   1 |  46 |   4 |
      [2] |   1 |   1 | 5976 |              [2] |   2 |   0 | 647 |
----------|-----|-----|------|------- ----------|-----|-----|-----|-------
                             | 99.06%                             | 98.33%
```

10-fold CV set 5:

```
class(t) = ^(IF(>=(+(+(^(90.826907|[1.350*Var16(t)])|+([1.350*Var16(t)]|
[1.000*Var3(t)]))|-(Cos([1.000*Var9(t)])|*([1.000*Var5(t)]|
[0.428*Var20(t)])))|^(^(+([1.000*Var2(t)]|98.567908)|Cos([-
0.239*Var16(t)]))|+(+([1.000*Var12(t)]|[1.000*Var4(t)])|+([1.000*Var6(t)]|1.584749))))|
ThenElse(+(-(+(-3.514523|[3.551*Var20(t)])|-([0.428*Var20(t)]|
[1.000*Var17(t)]))|IF(<=([1.813*Var7(t)]|[2.014*Var16(t)])|
ThenElse([0.442*Var2(t)]|60.781038)))|+(98.567908|-(-(1.584749|[-
0.239*Var16(t)])|^([1.000*Var16(t)]|1.584749)))))|Sig(+(^(+(+(
[2.496*Var20(t)]|0.000000)|+([-0.239*Var16(t)]|0.000000))|Cos(55.840663))|
*(Cos(-([1.000*Var9(t)]|[0.428*Var20(t)]))|
*(1.193392|+([1.350*Var16(t)]|[1.350*Var16(t)])))))))

Thresholds: [35.65; 79.5]

TRAINING                              TEST

Orig.  -> | [0] | [1] |  [2] |        Orig.  -> | [0] | [1] | [2] |
----------|-----|-----|------|        ----------|-----|-----|-----|
Est.  [0] | 145 |   4 |    5 |        Est.  [0] |  15 |   1 |   1 |
Class [1] |   5 | 326 |   29 |        Class [1] |   1 |  34 |   5 |
      [2] |   0 |   3 | 5963 |              [2] |   0 |   0 | 663 |
----------|-----|-----|------|------- ----------|-----|-----|-----|-------
                             | 99.29%                             | 98.89%
```

10-fold CV set 6:

```
class(t) = IF(<=(IF(<=(+([0.306*Var7(t)]|e^([1.000*Var12(t)]))|-(-
([4.358*Var20(t)]|0.587528)|Sqrt([0.500*Var16(t)])))|ThenElse(Cos(/
([1.000*Var4(t)]|[1.273*Var17(t)]))|[1.000*Var4(t)]))|IF(<=(+(
[7.775*Var20(t)]|+([1.273*Var17(t)]|[ 2.186*Var20(t)]))|+(-(0.000000|
-5.218301)|98.760572))|ThenElse([1.000*Var8(t)]|IF(>=(0.000000|
[1.000*Var2(t)])|ThenElse([0.306*Var7(t)]|[7.775*Var20(t)]))))))|
ThenElse(IF(<=(+([0.500*Var16(t)]| [1.000*Var12(t)])|-(-([4.358*Var20(t)]|
0.587528)|[1.000*Var12(t)]))|ThenElse(+(+(0.587528|98.760572)|0.587528)|
[1.000*Var12(t)]))|IF(<=(+([0.500*Var16(t)]|e^([2.353*Var16(t)]))|+(
Sqrt([1.000*Var4(t)])|15.374653))|ThenElse(+(+(0.587528|98.760572)|0.587528)|+(+(
[1.273*Var17(t)]| [2.186*Var20(t)])|/ ([3.649*Var18(t)]|9.638362))))))

Thresholds: [36.425; 80.1]

TRAINING                              TEST

Orig.  -> | [0] | [1] |  [2] |        Orig.  -> | [0] | [1] | [2] |
----------|-----|-----|------|        ----------|-----|-----|-----|
Est.  [0] | 145 |   9 |    4 |        Est.  [0] |  11 |   0 |   1 |
Class [1] |   8 | 325 |   26 |        Class [1] |   1 |  33 |   1 |
      [2] |   0 |   0 | 5963 |              [2] |   1 |   1 | 671 |
----------|-----|-----|------|------- ----------|-----|-----|-----|-------
                             | 99.27%                             | 99.31%
```

10-fold CV set 7:

```
class(t) = +(^(+(+(+(Log(8.775252)|+([1.440*Var20(t)]|8.775252))|*(Sin(
[0.158* Var20(t)])|-(8.775252|[1.000*Var16(t)])))|e^(Sin(+([0.158*Var20(t)]|
[1.000*Var16(t)]))))|Sig(+(+(Sin([-0.421*Var16(t)])|+([0.086*Var4(t)]|
[1.285*Var20(t)]))|+(+([-1.893*Var16(t)]|[1.000*Var8(t)])|
Sin([1.000*Var16(t)]))))))|IF(<=(Sig(+(-(59.824802|[1.000*Var2(t)])|-
(8.775252|[1.000*Var19(t)])))|+(Log(+([1.000*Var2(t)]|8.775252))|+(e^(
[1.000*Var7(t)])|+([-1.893*Var16(t)]|0.000000))))|ThenElse(+(59.824802|
-(2.810579|[0.158*Var20(t)]))|+(*(*([0.249*Var2(t)]|[0.158*Var20(t)])|
[0.158*Var20(t)])|+(*([0.158*Var20(t)]|[0.158*Var20(t)])|
+([0.086*Var4(t)]|[1.000*Var17(t)]))))))

Thresholds: [41.575; 74.575]

TRAINING                              TEST

Orig.  -> | [0] | [1] |  [2] |        Orig.  -> | [0] | [1] | [2] |
----------|-----|-----|------|        ----------|-----|-----|-----|
Est.  [0] | 140 |  11 |    5 |        Est.  [0] |   9 |   3 |   3 |
Class [1] |  13 | 305 |   30 |        Class [1] |   3 |  49 |   4 |
      [2] |   1 |   0 | 5975 |              [2] |   0 |   0 | 649 |
----------|-----|-----|------|------- ----------|-----|-----|-----|-------
                             | 99.07%                             | 98.19%
```

10-fold CV set 8:

```
class(t) =
-(IF(<=(-(+(+([5.748*Var20(t)]|[1.000*Var4(t)])|3.807626)|*(
-(1.928774|[1.000*Var7(t)])|-([3.224*Var16(t)]|[1.000*Var8(t)])))|
IF(<=(-([3.351*Var16(t)]|[1.000*Var2(t)])|*(1.000000|3.807626))|
ThenElse([-5.157*Var20(t)]|+([4.698*Var16(t)]|56.916803))))|
ThenElse(-(+(+([1.000*Var16(t)]|-8.636405)|+([5.518*Var16(t)]|-8.636405))|-(-([-
5.157*Var20(t)]|[1.000*Var16(t)])|e^([0.222*Var17(t)])))|
IF(<=(-([3.351*Var16(t)]|[1.000*Var7(t)])|*(1.000000|3.807626))|
ThenElse(+([5.518*Var16(t)]|102.747052)|+([1.000*Var20(t)]|56.916803)))))|IF(>=(-
(3.807626|+(+([1.000*Var15(t)]|[1.000*Var11(t)])|+([1.000*Var7(t)]|
[-5.157*Var20(t)])))|Log(/(e^(56.916803)|Sqrt([4.697*Var2(t)]))))|
ThenElse(-(-(3.807626|*([3.224*Var16(t)]|5.880540))|Cos(+([4.698*Var16(t)]
|1.000000)))|*(/(+(3.464115|[1.000*Var4(t)])|+([1.000*Var4(t)]|1.928774))|+(
[4.698*Var16(t)]|Sig([5.748*Var20(t)]))))))

Thresholds: [36.375; 83.275]

TRAINING                              TEST

Orig.  -> | [0] | [1] |  [2] |        Orig.  -> | [0] | [1] | [2] |
----------|-----|-----|------|        ----------|-----|-----|-----|
Est.  [0] | 143 |  11 |    3 |        Est.  [0] |  16 |   1 |   3 |
Class [1] |   6 | 317 |   33 |        Class [1] |   0 |  38 |   6 |
      [2] |   1 |   1 | 5965 |              [2] |   0 |   0 | 656 |
----------|-----|-----|------|------- ----------|-----|-----|-----|-------
                             |  99.15%                            | 98.61%
```

10-fold CV set 9:

```
class(t) = +(+(IF(>=(^(Log([3.598*Var20(t)])|+([2.767*Var7(t)]|[1.000*Var12(t)]))|
Log([2.368*Var16(t)]))|ThenElse(IF(>=([2.368*Var16(t)]|35.377930)|
ThenElse(-21.656405|28.887860))|+(+([1.000*Var17(t)]|-3.329153)|+ ([0.247*Var2(t)]|-
3.329153))))|IF(>=(+(+([1.000*Var6(t)]|28.887860)|+
([2.368*Var16(t)]|[1.000*Var10(t)]))|*(-([0.861*Var16(t)]|[1.000*Var3(t)])
|[1.000*Var18(t)]))|ThenElse(IF(>=([2.368*Var16(t)]|49.112689)|ThenElse(
22.857204|49.112689))|-([3.484*Var18(t)]|+([1.000*Var18(t)]|-5.595520)))))|+(IF(<=(-
([2.368*Var16(t)]|/(1.182745|[1.000*Var17(t)]))|
+([1.000*Var8(t)]|[1.000*Var17(t)]))|ThenElse(Sin(Sig(-21.656405))|
IF(<=(35.377930|[3.722*Var20(t)])|ThenElse([0.894*Var16(t)]|-21.656405))))|IF(>=(^(-
([1.000*Var18(t)]|[1.000*Var2(t)])|+(
[2.279*Var7(t)]|[1.000*Var4(t)]))|Log([2.368*Var16(t)]))|ThenElse(IF(>=([0.861*Var16(t)]|
22.857204)|ThenElse([2.279*Var7(t)]|22.857204))|IF(<=(
35.377930|[3.598*Var20(t)])|ThenElse([0.247*Var2(t)]|-21.656405))))))

Thresholds: [34.225; 75.6]

TRAINING                              TEST

Orig.  -> | [0] | [1] |  [2] |        Orig.  -> | [0] | [1] | [2] |
----------|-----|-----|------|        ----------|-----|-----|-----|
Est.  [0] | 145 |   0 |    3 |        Est.  [0] |  17 |   0 |   1 |
Class [1] |   3 | 341 |   21 |        Class [1] |   1 |  27 |   6 |
      [2] |   0 |   0 | 5967 |              [2] |   0 |   0 | 668 |
----------|-----|-----|------|------- ----------|-----|-----|-----|-------
                             | 99.58%                             | 98.89%
```

Figure 15.6: Example classification models (produced using GP) and the resulting confusion matrices for the *Thyroid* data set.

15.2 Quality Pre-Assessment in Steel Industry Using Data Based Estimators

In this section we give a summary of the research done on quality pre-assessment in steel industry. The goal of this work was to examine the ability of data based estimators to formulate models that predict the final quality of steel products on the basis of process parameter values.
The work described here was done during a research project at the Institute for Design and Control of Mechatronical Systems at Johannes Kepler University Linz, Austria, in cooperation with the Industrial Competence Center for Metallurgical Process Engineering, Austria and has been partly funded by the Austrian Ministry for Economy and Labor in the frame of its Industrial Competence Center Program K-ind/K-net. I am thankful to Prof. Dr. Luigi del Re and Dipl.-Ing. Hajrudin Efendic, who managed this project and helped to obtain and analyze the results presented here. An extended version of the report given here has already been published in [WEdR06].

15.2.1 Introduction

Quality assessment is a standard and central issue in industrial processes and is usually performed on the basis of an inspection of the final product, either by a human operator or in a computer assisted way. The latter approach includes usually an automatic inspection and/or classification method, very often based on pattern recognition tools. A much more appealing possibility, however, consists in performing an indirect assessment, i.e. without visual inspection of the final product. These methods can include intermediate process data and are therefore not necessarily predictive in a strict sense, but offer the essential advantage of allowing to understand the relationships between process quantities and quality. To this end, different approaches can be used, in particular a classical issue is the choice or combination of model based vs. data based approaches.

We here partially summarize a case study within which purely data based approaches were used to predict the quality of steel products, where the results of the human inspection were used as comparison, partly as training and partly as validation data. Essentially, the experience can be summarized as follows:

- It is indeed possible to implement an automatic quality assessment scheme which reproduces rather well the results of the human inspection of the final product.

- The specific choice of the modeling algorithms, as long as nonlinear modeling methods are used, is not the critical issue.

- The binary decision on the quality by the human operator is the crisp expression of a continuous value, and therefore is the wrong quantity on which to train the algorithms.

- Even though large amounts of data are recorded on typical steel (and other) industrial plants, they might not contain the necessary information.

15.2.2 Solution Structure

As already stated, the first experience gained in this test case concerns the fact that it is indeed rather straightforward to implement an automatic quality assessment scheme which allows a good forecast of the product quality. Even if this affirmation could be obvious for many different setups, it is not for steel plants because the data are poor in some respects:

- All data available arise from real production processes, and, clearly, the operator tries to keep the process values in ranges known to produce “best” results, thus the diversity of data is very limited.

- Luckily enough, the number of products classified as faulty is much lower than the number of products classified as correct, but this means that the number of experiments available for the faulty class is much lower than for the correct class.

Under these circumstances, the simple use of standard classifiers directly is likely to lead to ill-conditioned problems, whose solution may yield good results for the training phase, but will usually perform very poorly for new or validation data. To cope with such a setup, the general structure schematically presented in Figure 15.7 can be used. This structure is intended to derive an adjacent, better conditioned problem whose solution is expected to behave better - on average - both in the training and in the validation set.

The first part of the overall data processing workflow is the preprocessing of the data. This preprocessing includes the removal of variables without significant

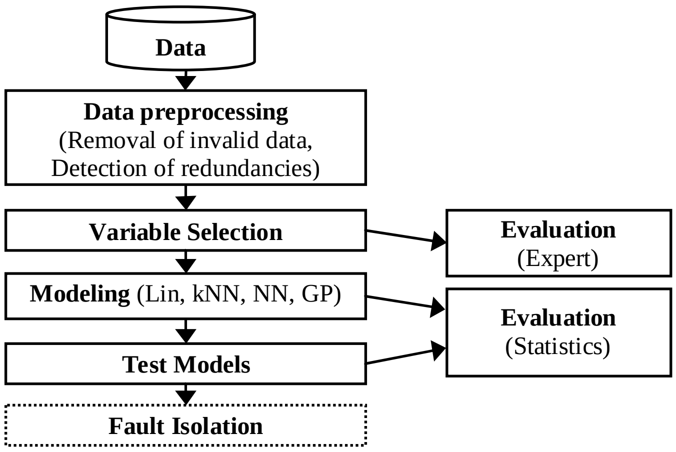


Figure 15.7: The general steel production data processing framework.

information and the detection of redundancies (variable-wise as well as sample-wise). The next issue is the selection of variables that seem to have a statistically measurable relationship with the target variable (in the case of our project the product quality). Then, models are created to describe the target variable's values in terms of the values of the other retained variables. Finally, the model's quality is calculated using the part of the data that was not included in the training data base.

15.2.2.1 Preprocessing

The purpose of preprocessing is the extraction of the informative data from the signals. In our setup there are no dynamical data, so that this operation boils down to the elimination of all variables not storing significant information and to the identification of redundancies within the data. The purpose of the latter operation is essentially to reduce the problem complexity.

Redundancy detection is in fact a bivariate problem. The detection of redundant variables can be very meaningful also in view of a possible fault detection and isolation (FDI), and is typically based on pair-wise correlation coefficients (between the full ordered value sets of the respective variables).

The detection of redundant samples works quite differently, the goal being to find samples (formed by points of two or more variables belonging to the same measurement) with "insignificant" (variable-wise) differences. It is essentially a clustering approach, in which all elements which belong to a sufficiently small environment are considered as a single measurement. Clustering techniques are well known; in this work, all variables have been normalized independently, the distance of any sample to any other has been calculated as a mean squared error, and a threshold has been used to filter out redundant samples. Both these redundancy checks have proven to be quite important to obtain a feasible problem. Still, any "approximate" redundancy can indeed hide a real difference. In practice, this means that the value of the threshold cannot be computed a priori but must be determined during the training phase. It has also turned out important to perform the sample-wise redundancy analysis for all samples of each given class separately, i.e., the classification variable (for example the one dividing all samples into the "faulty" and the "not faulty" ones) should not be considered in calculating the distance matrix. This enables finding the pairs of really redundant samples.

The analysis of the data, however, has shown the existence of a special class of "redundant" cases: Samples clearly redundant in all process parameters but with a different classification. A more precise analysis of such cases has shown that they corresponded to a "critical" chemical composition known to produce not well repeatable results. Such cases can also be seen as indicators that there could be additional factors which have relevant influence on the process outcome, but are not measured.

15.2.2.2 Variables Selection

The objective of variable selection is to identify from measurement data a list of variables which are related; the aim is to derive a set of "input" variables $u_{i,1}, u_{i,2}, \ldots, u_{i,m}$ which significantly influence a defined target variable y_i. The so obtained list allows exploring functional relations between different process variables, which may extend expert knowledge, and provide a set of input signals which a model should be built with. There are different methods known for variable selection as for example exhaustive search, sequential forward selection, or sequential backward selection.
Exhaustive search is executed by computing all possible combinations of variables and evaluating them by means of the sum of squared errors; exactly that combination of variables will be selected which provides best approximation of measurement data. This method is able to provide an optimal solution (if the process is linear), but especially for higher dimensional problems (including big numbers of variables) it requires excessive computation time. In order to overcome this drawback, forward and backward selection can be used as alternatives even if they provide only sub-optimal solutions.

Sequential forward selection assumes a linear-in-parameters model of the form

$$y_i = \phi^T(x_{i,1}, x_{i,2}, \dots, x_{i,n}) * \theta \qquad (15.17)$$

where T denotes a vector of known basis functions (linear, polynomial, etc) and θ the unknown parameter vector. The algorithm sequentially derives the list of input variables. In the first step, only one input variable is considered where that variable is selected that minimizes the sum of squared errors. In the next step, another input variable is selected where once again that variable is chosen which minimizes the sum of squared errors; the algorithm iteratively adds more and more input signals to the set X until a predefined accuracy is reached and hence the algorithm terminates. Of course the results depend on the chosen basis functions; different models (linear and polynomial) were used in this project and the largest common set $X = \bigcap_{i=1\dots m} X_i$ was computed. The main difference when applying backward selection is that the algorithm starts with all variables available in a set of selected variables and then iteratively removes variables that do not have a statistically measurable connection with the observed (measured) target values. Hybrid variants combining backward selection and a subsequent forward selection step have also been investigated for producing good results very efficiently.

15.2.2.3 Modeling and Model Based Classification

There are several modeling methods that can be used in the context of estimating the quality of steel products; in the following we report on our experience using linear modeling, k-nearest-neighbor (kNN) classification, neural networks (NN) and genetic programming (GP).
In addition to building mathematical models, thresholds separating the classes '0' and '1' have to be found. Of course, thresholds are fixed on the basis of the evaluation of the models on the training data; they are set so that as few misclassifications as possible are recorded. During our intensive test series we have come to the conclusion that weighting factors 1:5 seem to be appropriate (i.e., when it comes to fixing thresholds, misclassifications of faulty samples are weighted 5 times as much as misclassified fault free samples).

Linear modeling and neural networks are used in exactly the same way as described in Section 14.3; GP is applied as described in the first part of this thesis, especially using gender specific parents selection (random and proportional, see Section 4.1), offspring selection (Section 4.2), local optimization and pruning (Chapter 10), and classification specific evaluation and results analysis as described in Section 8.2.

Unlike other data based modeling methods, kNN classification works without creating any explicit models. During the training phase, the data are simply collected; when it comes to classifying a new, unknown sample x_{new}, the sample-wise distance between x_{new} and all other training samples is calculated and the classification is done on the basis of those k training samples showing the smallest distances from x_{new}. There is a lot of literature that can be found for kNN classification; very good explanations and compact overviews of kNN classification are for example given in [DHS00] and [RN03].

15.2.3 Empirical Results

In this section we exemplarily summarize the results achieved using the data processing methods discussed previously for analyzing data from a large scale industrial production plant and the binary information about the respective final products' quality. For confidentiality reasons, the variables' names have been substituted by dummies, and the results of the redundancies check methods cannot be discussed in detail. Still, there are some issues to be discussed regarding these results.
Within our testing series we have executed a so-called 10-fold cross validation (10-fold CV) series using the data (consisting of 2400 samples and 85 variables, one of them indicating the resulting fault occurrences) taken from a real world steel production plant already mentioned before. I.e., the given data were split into 10 commensurate independent sets of samples yielding 10 pairs of test data (each containing 10% of the given data) and training data (each including the remaining 90%, respectively).

15.2.3.1 Redundancies and Nearest-Neighbor Analysis Results

In general, measurement systems of industrial plants include redundant samples as well as redundant signals, for instance due to multiple sensors measuring the same or very similar quantities; this happened also here.
Redundant samples are also to be expected, simply because parameter settings of production plants are usually not changed randomly after each production process, at least if the earlier run was successful. As already explained, however, sometimes samples redundant in the inputs correspond to different outcomes: A clear sign that something is wrong or missing here. Of course, this proposition has to be examined statistically. Since the detection of redundant samples depends on the choice of the respective threshold, it is not easy to objectively estimate the ratio of redundant samples that show this anomaly. Thus we have executed a simplified

nearest neighbor analysis on all training data available: For each sample, the nearest neighbor sample was identified and then we analyzed, how many sample of each class have nearest neighbors showing a different original classification. As one can see in Table 15.10 that summarizes this analysis, almost half of the samples originally belonging to class '1' have nearest neighbors that are classified differently.

Table 15.10: Simplified nearest neighbor analysis for steel production data.

Original	*Nearest Neighbor Classification*	
Classification	Class '0'	Class '1'
'0'	97.13%	2.87%
'1'	43.45%	56.55%

15.2.3.2 Variables Selection Results

Apart from using variable selection as a data processing and reduction step, the sets of selected variables can also be used as an independent data analysis result. Even though these results are not to be seen as descriptions of causal dependencies, this information can be very useful for experts analyzing the system investigated since it can be used for describing relationships between certain signals and the final product quality (even though these relationships cannot be quantified without further analysis). Especially the graphical representation of these relationships and redundancies has in the past already been a very appropriate basis for further discussions and analysis of complex industrial production plants.

Graphical as well as tabular information about the sets of relevant variables detected using different variables selection methods is given in [WEdR06].

15.2.3.3 Modeling Results

One of the most obvious results of our test series was that linear models are not adequate for modeling the relationship between steel production process variables and the respective final products' quality. kNN classification also does not seem to be the ideal method; even though class '0' was here almost always classified correctly, the ratio of correct classifications of '1' is very low. This is probably related to the fact that fault-free experiments are represented in the data much more often (approx. 90%) than faulty productions; fault-free samples therefore tend to be "near" also to faults parameter-vectors.

As described in more detail in [WEdR06], the data available were split into training and test data, and then variables selection (except when using GP) and the modeling approached listed above were applied. Table 15.11 summarizes the modeling results in terms of correct classification rates for each class on training as well as on test data.

Table 15.11: Overview of the modeling results; for each method, the ratio of correct classifications is given.

Method	*Training Results*		*Test Results*	
	Class '0'	*Class '1'*	*Class '0'*	*Class '1'*
Lin	76.59%	91.18%	76.25%	87.39%
kNN	–	–	97.84%	49.53%
NN	87.33%	95.59%	84.34%	82.83%
GP	89.18%	68.55%	89.85%	73.33%

The results achieved using NNs and GP are worth a closer look; both methods have yielded more or less acceptable results, but still there are some significant differences that are to be discussed in the following. NN classifiers show a rather high correct classification rate on training data, correctly classifying more than 87% of class '0' and even more than 95% of the samples belonging to class '1'. Nevertheless it is obvious that overfitting has happened here because the correct classification rates on test data are much lower.

Even though the classification models produced by GP show a slightly worse performance, there are some other important details to be considered here. The most important fact is that the GP algorithm was set to produce rather simple, interpretable models; an exemplary formula is graphically shown in Figure 15.8. For sure it would be possible to achieve higher classification rates by increasing the allowed model complexity; this, of course, would in return trigger the production of formulae that are not easy to interpret any more. Furthermore, GP seems to be not as exposed to overfitting problems as NNs; this is why the classification rates using the formulae produced by GP on test samples are not significantly worse than those achieved on training data. A rough overview of the modeling results is given in Table 15.11.

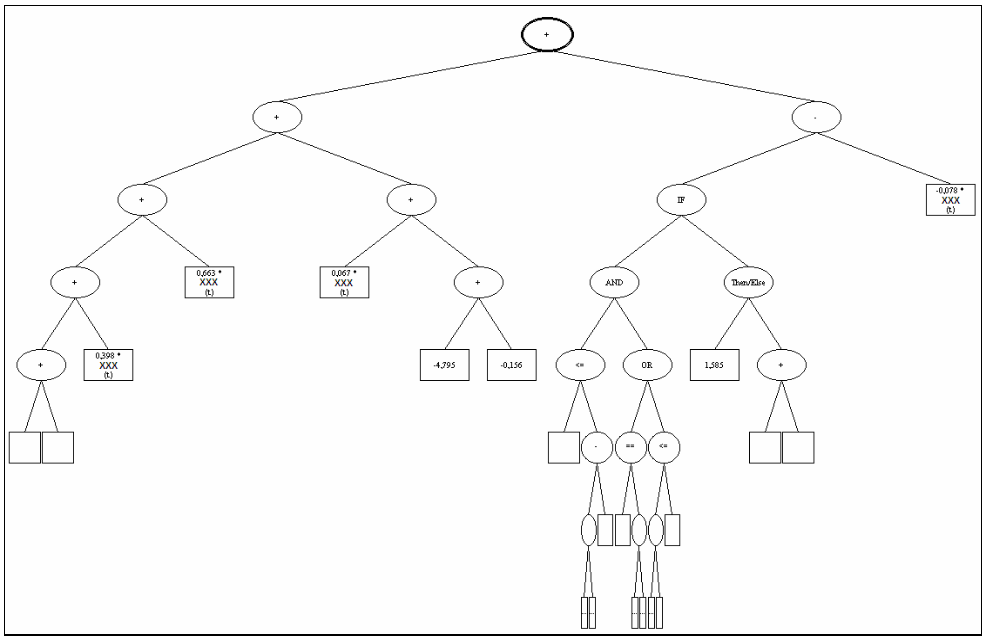


Figure 15.8: Model identified by the GP based classification algorithm using the first 2001 samples of the steel production data set.

15.2.4 Discussion

As stated in the beginning, the method can be easily automated and the result - approximately 90% correct classification - would be considered quite satisfying for many applications. In the case of a production plant, however, it is evident that 90% is still too little, a visual control of a significant part of the products remains necessary. Even though some improvements could be still possible by tuning the methods applied, the experiments have clearly shown that this is not really the issue. Essentially, there are some important issues to be considered:

- The analyzed process is (partly) stochastic, i.e. the reproducibility is limited.
- The stochastic aspect is increased by the fact that the classification is performed by human operator(s), whose classification thresholds might vary over time and between persons.
- The recorded data do not need to contain all the necessary information.

Against this background, a real improvement is not expected by modifications of the algorithms alone, but by a change in the problem. This can be achieved on two fronts:

- Instead of predicting the outcome of the classification, the data analysis tool should be used to give estimations on the likelihood of the product to be faulty.
- The data set should be checked for sufficient information content for producing nonlinear models with sufficient degrees of prediction precision. This is approximately the case when the ratio between parameters and independent samples is large enough (typically more than 10). Otherwise, the measurement system has to be extended (either by measurements, by models or by expert knowledge).

Please notice that this implies a different classification procedure (at least three classes are needed, namely “faulty”, “correct” and “undetermined”), but ideally a quantitative assessment of the degree of fault would be much more appropriate. Still, going from crisp to probability values could be an asset if the data analysis tool is to be used for process re-design and/or optimization. If these aspects are considered, we are confident that a very valuable tool can be designed to predict the product quality. If the tool is then used to operate the plant under correct conditions, i.e. producing as many correct outputs as possible, this will reinforce the quality of the prediction tool and ask for very few visual control steps. These, however, will remain as long as the stochastic nature of the process subsists.

Chapter 16

GP in Volatile Environments: On-Line and Sliding Window GP

16.1 Simulated On-Line Design of Virtual Sensors

Early test series using the simulated on-line GP approach described in Section 13.1 have been originally published in [WEA+05], [WAW05a] and later in [WEA+06]; a compact overview of these results is given here. In fact, this approach was not actually used in an on-line identification context; we simulated an on-line identification scenario using data that were recorded previously.

Test Environment

For testing the presented on-line learning GP algorithm we have analyzed the data representing several signals of a BMW M47D diesel engine (with activated exhaust recirculation as described in Section 14.1.1). Again, the goal was to identify a model for the engine's NO_x emissions using the measured values of several other engine parameters. A whole FTP 75 cycle was executed within approximately 1,400 seconds; all sensor signals (in total 33) were recorded with 20 Hz resolution, for the GP identification algorithm the data was downsampled to 5 Hz resolution. For simulating an on-line learning scenario, initially only 50 samples are inserted into the algorithm's data pool adding one more every 0.2 seconds. Since the data basis available to the identification algorithm grows constantly during the simulation (which brings along runtime problems), the identification data was restricted to the most

recent 500 samples (representing 100 seconds).
As underlying GP algorithm the SASEGASA was applied working with a population size of 300 individuals, 5% mutation rate and a combination of random selection and roulette selection as selection operators with generational replacement. The average of squared errors was chosen as fitness function with activated early stopping as described in Section 7.5.5; after each 10% of the given training data the stopping criterion was checked and the evaluation aborted as soon as it was clear that the solution's fitness was going to be worse than the parents' qualities.

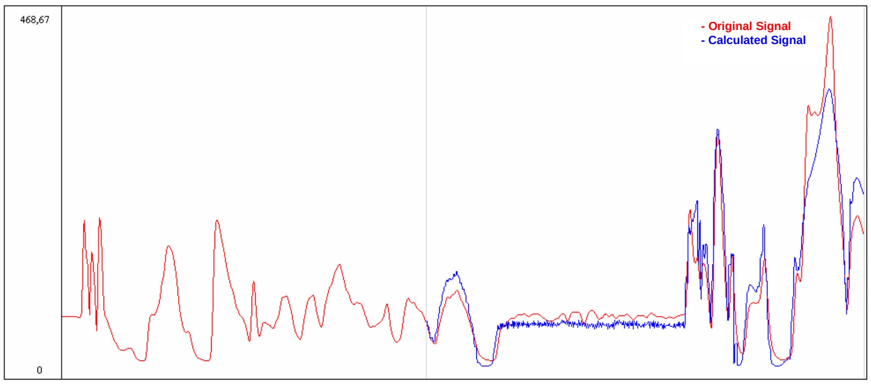


Figure 16.1: Evaluation of the best result after three minutes.

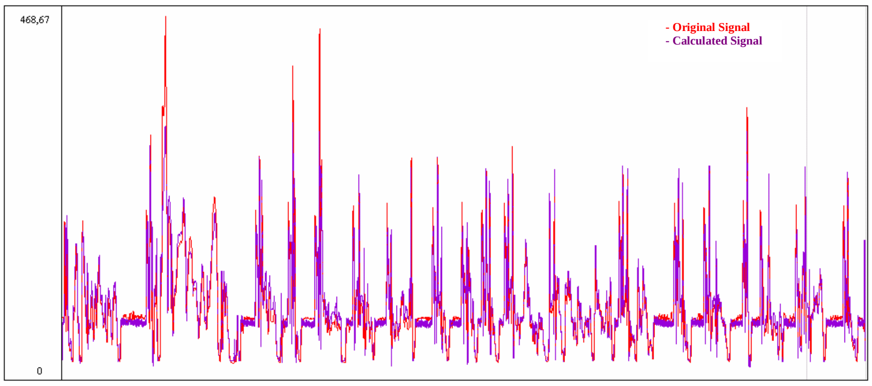


Figure 16.2: Evaluation of the best result after end of the FTP cycle.

Test Results

The Figures 16.1 and 16.2 illustrate the algorithm's behavior and graphically show evaluations of the currently best models after some minutes (Figure 16.1) and at

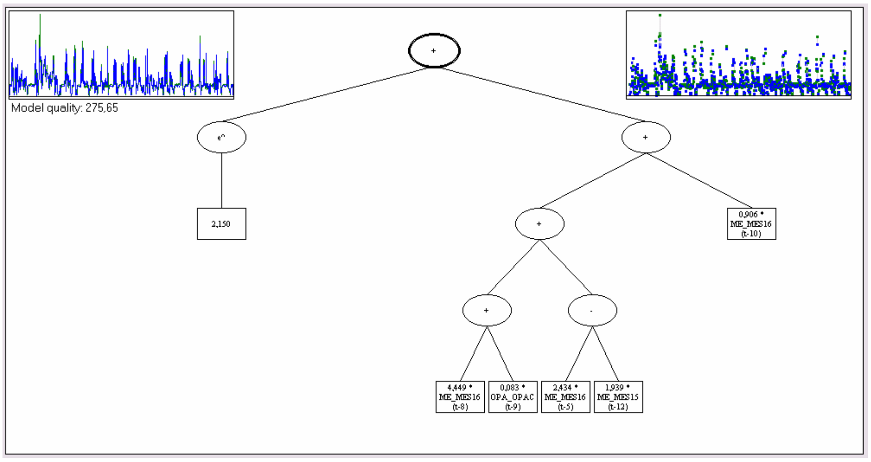


Figure 16.3: Model identified by on-line GP after the whole FTP cycle.

the end of the whole simulation (Figure 16.2). The model that was returned by the program in the end (after finishing the whole simulation, i.e. after approximately 23 minutes), is shown in Figure 16.3. The input variables of this model for NO_x are the target quantity of the fuel injection pump ME_MES16, the opacity of the engine's emissions OPA_OPAC and the starting time of the fuel injection ME_MES15 with varying coefficients and time offsets. In other words, the engine's NO$_x$ emissions can be modeled as

$$NO_x \sim f(ME\_MES16, OPA\_OPAC, ME_MES15). \qquad (16.1)$$

This result was checked and rated as very good by experts in the field of automotive control, namely members of the automotive group of the Institute of Design and Control of Mechatronical Systems at the University of Linz, Austria. It is in fact also consistent with those retrieved during previous investigations ([dRLF$^+$05], [AdRWL05]).

In addition to the test run documented above we have tested the same data set several times applying the same algorithmic parameter settings. These test runs were all executed independently and produced also structurally different formulae modeling the engine's NO$_x$ emissions. One of these models is graphically shown in Figure 16.4; NOx is modeled using the variables ME_MES16, the fuel consumption KW_VAL and the temperature of the coolant TWA, again with varying coefficients and time offsets. I.e., the engine's NO$_x$ emissions can also be modeled as

$$NO_x \sim f(ME\_MES16, KW\_VAL, TWA). \qquad (16.2)$$

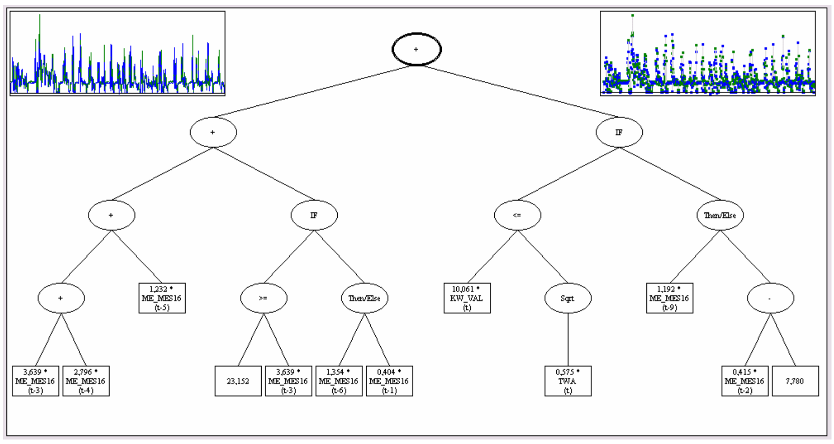


Figure 16.4: Alternative model identified by on-line GP after the whole FTP cycle.

Even though this model is not quite as good as the one we have stated previously (evaluated on the whole test data set its average squared residual is approximately 16% higher), it can still be used for fault detection because it gives a good approximation of the original target values and is consistent with the results retrieved during previous investigations. Due to the fact that its set of input signals differs from the set of inputs of the model previously mentioned, these two models can be used for data-based fault diagnosis based on analytical redundancy. For a detailed explanation of fault detection, fault isolation and the potential role of GP in this context please see for example [WEA$^+$05] or [WEA$^+$06] and the references given therein.

So, on the basis of evolution inspired heuristic optimization techniques, an enhanced on-line learning and model structure identification approach based on Genetic Programming has been presented and successfully tested on real-world measurement data.

16.2 Selection Pressure Based Sliding Window GP

We shall now discuss empirical test results obtained using the selection pressure driven sliding window approach for GP as described in Section 13.2; the results summarized here have been previously published in [WAW07b]. Here we again report on tests executed using the *Thyroid* data set; we are going to report on test results achieved using the first 80% of the data (containing 7,200 samples in total) as training data and the remaining 20% for testing the models created. We here state the quality of the classifiers created by the identification process using the mean squared error function for evaluating them.

16.2.1 Parameter Settings and Test Results

For testing the sliding window approach presented here and also for comparing its ability to produce models of high quality we have tested the following 5 different GP-based data mining strategies characterized by their population size $|pop|$, maximum selection pressure values MSP (maximum selection pressure), $MSP1$ and $MSP2$ (maximum selection pressure values 1 and 2 as explained in Section 13.2) and relative values for sliding window parameters:

1. Standard-GP: $|pop| = 2000$, 1500 generations, no offspring selection.
2. GP including offspring selection: $|pop| = 1000$, $MSP = 200$
3. Sliding window GP: $|pop| = 1000$, $MSP1 = 50$, $MSP2 = 200$, sliding window: initial size 0.2, step width 0.1, maximum size 0.4
4. Sliding window GP: $|pop| = 1000$, $MSP1 = 50$, $MSP2 = 200$, sliding window: initial size 0.4, step width 0.2, maximum size 0.5
5. Sliding window GP: $|pop| = 1000$, $MSP1 = 20$, $MSP2 = 200$, sliding window: initial size 0.2, step width 0.05, maximum size 0.4

All tests were executed applying 15% mutation rate and a combination of random and roulette parent selection schemata. For each test scenario we have executed 5 independent test runs. In the following table we give average numbers of iterations and solutions evaluated as well as the quality of the models identified (average values as well as the quality of each test series' result showing the best fit on the complete training data set) with respect to (complete) training and test data. Please note that all variables were normalized independently (i.e. scaled linearly so that the resulting variables' mean values are equal to 0 and their standard deviations are exactly 1.0). The maximum size of models created by the training algorithm was set to 60, the maximum formula tree height to 8.

In Figure 16.5 we give two characteristic screenshots: In the left part the selection pressure progress of one of the test runs of test series (5) is displayed (with vertical

Table 16.1: Results of the tests executed for the *Thyroid* data set.

Test scenario	Iterations (average)	Solutions evaluated (average)	Speed up	Model quality (mean squared error) on training data: Best model	on training data: Average	on test data: Best model	on test data: Average
1	1,500.00	3,000,000.00	1.00	0.316	0.410	0.381	0.444
2	64.40	2,717,678.80	1.10	0.155	0.283	0.251	0.341
3	65.60	3,199,551.00	2.40	0.166	0.193	0.219	0.233
4	59.80	1,925,755.40	3.18	0.166	0.246	0.199	0.310
5	62.60	2,483,116.60	3.10	0.125	0.173	0.220	0.252

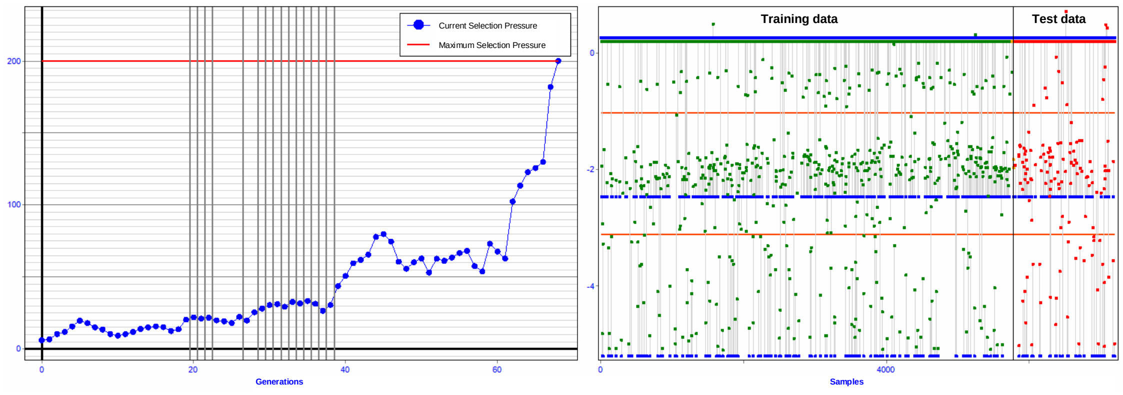


Figure 16.5: Left: Selection pressure progress of the best test run of test series (5); Right: Graphical representation of the best test run of test series (4).

gray lines indicating training data scope drifts: every time the selection pressure became greater than 20, the window was shifted); in the right part we show a graphical representation of the evaluation of the best classifier trained in test series (4). This model correctly classifies 98.46% of the training and 98.08% of the test samples; a detailed confusion matrix is given in Table 16.2.

Obviously, as is summarized in Table 16.1, all GP methods using offspring selection perform significantly better than the standard implementation. Furthermore, the use of sliding window mechanisms here resulted in models that perform better on test data as well as in significant runtime reduction. Due to the fact that on the one hand not all training data but only the respective current data scopes are used in the sliding window test series and on the other hand the share of model evaluation in runtime consumption of GP based data mining is almost 100%, the algorithms are executed significantly faster: The respective speed up values range from 2.4 to almost 3.2.

Table 16.2: Analysis of the best model produced in test series (4) whose evaluation is displayed in Figure 16.5.

Original Class →		1	2	3	
Class Predicted	1	29 (2.06%)	0 (0.00%)	2 (0.14%)	
	2	5 (0.35%)	63 (4.47%)	19 (1.35%)	
	3	0 (0.00%)	1 (0.07%)	1290 (91.55%)	
Correctly Classified					1382 (98.08%)

16.2.2 Discussion

In this section we have summarized the results of sliding window tests for data mining using GP. Offspring selection is used for determining the resulting selection pressure which is used for triggering the drift of the current training data scope; we have reported on a series of tests using a widely used classification benchmark problem for demonstrating the effects of the use of these enhanced aspects. It has been shown that it is possible to reduce the algorithm's runtime as well as to increase the models' test quality when applying the sliding window mechanism presented here.

Still, more detailed analysis of this method is needed. For example, the effects of this drifting mechanism on the genetic diversity are to be analyzed; we are going to report on this analysis in Chapter 17.

Chapter 17

Population Dynamics

17.1 Genetic Propagation

17.1.1 Test Setup

When speaking of analysis of genetic propagation as described in Section 12.1, we analyze how well which parts of the population succeed in propagating their genetic material to the next generation, i.e. to produce offspring that will be included in the next generation's population. In this section we shall report on tests in this area; major parts have been published in our article on offspring selection and its effects on genetic propagation in GP based system identification [WAW08a].

We have here used the NO_x data set already presented and described in Section 14.2.1. Originally, this data set includes 10 variables, each storing approximately 36,000 samples; the first 10,000 samples are neglected in the tests reported on here, approximately 18,000 samples are training and 4,000 samples are validation (which is in this case equivalent to test) data. The last ∼4,000 samples are again neglected.

In principle, we are using conventional GP (with tournament and proportional selection) as well as extended GP (with gender specific selection as well as offspring selection). The details of the test strategies used are given in Table 17.1.

In all three test strategies we applied subtree exchange crossover, the time series analysis specific evaluation function (with early abortion as described in Section 8.1.3) for evaluating solutions, and 1-elitism as well as 15% mutation rate.

Table 17.1: GP test strategies.

Strategy	*Properties*
(I) *Conventional GP*	|Pop| = 1000; Tournament parents selection ($k = 3$) nr. of rounds: 1000
(II) *Conventional GP*	|Pop| = 1000; Proportional parents selection; nr. of rounds: 1000
(III) *Extended GP*	|Pop| = 500; Gender specific parents selection (proportional, random); Offspring selection (SuccessRatio = 1, MaxSelPres = 100)

17.1.2 Test Results

We have executed independent test series with 5 executions for each test strategy; the results are to be summarized and analyzed here.

With respect to solution quality and effort[1], the extended GP algorithm clearly outperforms the conventional GP variants (as summarized in Table 17.2).

Table 17.2: Test results.

		I	*II*	*III*
Best Quality (Training)	min.	1,390.21	3,022.12	1,201.23
	avg.	1,513.84	5,014.96	1,481.69
	max.	2,431.54	10,013.12	2,012.27
Best Quality (Test)	min.	8,231.76	12,312.83	4,531.56
	avg.	10,351.96	15,747.69	8,912.61
	max.	13,945.23	21,315.23	16,123.34
Generations		500		64.31
Effort		1,000,000		898,332.23

Regarding parents analysis, in all test runs we documented the propagation count

[1]The number of solutions evaluated is here interpreted as the algorithm's total effort.

for each individual and sum these over all generations. So we get

$$pc_{total}(i) = \sum_{i \in [1;gen]} pc(i) \qquad (17.1)$$

for each individual index i and assuming that gen is the number of generations executed. Additionally, we form equally sized partitions of the population indices and sum up the pc_{total} values for each partition.

In Table 17.3 we give the average pc_{total} values for percentiles of the populations of test series I, II and III; for test series I and II we collected the pc_{total} of 100 indices for forming a partition, for test series III we collected 50 indices for each partition. The Figures 17.1 and 17.2 show pc_{total} values of exemplary test runs of the series I and II summed up for partitions of 10 solution indices each, Figure 17.3 shows pc_{total} values of exemplary test runs of series III summed up for partitions of 5 solution indices each.

Table 17.3: Average overall genetic propagation of population partitions.

Population	*Test Strategy*		
Percentile	*I*	*II*	*III*
0	27.88%	10.31%	13.54%
1	21.29%	10.35%	11.20%
2	16.65%	10.31%	11.67%
3	12.64%	10.26%	10.91%
4	9.06%	10.25%	10.63%
5	6.08%	10.28%	9.85%
6	3.71%	10.24%	9.39%
7	1.88%	10.16%	8.83%
8	0.72%	10.10%	7.92%
9	0.10%	7.74%	6.07%

As we see from the results given in Tables 17.2 and 17.3 and Figure 17.1, there is a rather high selection pressure when using tournament selection; the results are rather good and (as expected) less fit individuals are by far not able to contribute to the population as well as fitter ones, leading to a quick and drastic reduction of genetic diversity.

The results for test series II, as given in Tables 17.2 and 17.3 and Figure 17.2, are significantly different: The results are a lot worse (especially on training data) than those of algorithm variant I, and obviously there is no strong selection pressure

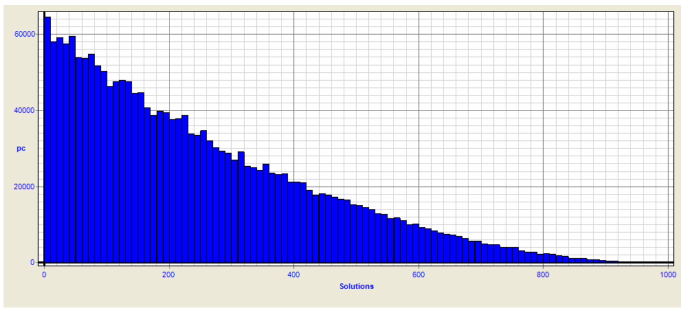


Figure 17.1: pc_{total} values for an exemplary run of series I.

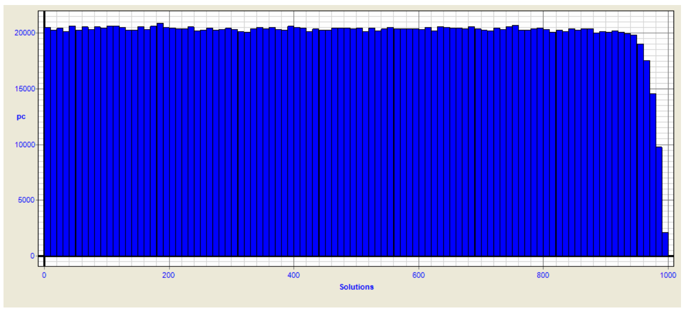


Figure 17.2: pc_{total} values for an exemplary run of series II.

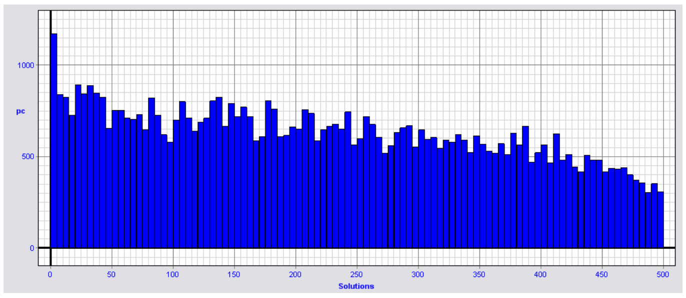


Figure 17.3: pc_{total} values for an exemplary run of series III.

as almost all individuals (or, rather the individuals at the respective indices) are able to contribute almost to the same extent. Only the worst ones are not able to propagate their genetic material to the next generations as well as better ones. This is due to the fact that in the presence of very bad individuals roulette wheel selection selects the best individuals approximately as often as those that perform middlingly well. Especially in data based modeling there are often individuals that score extremely badly (due to divisions by very small values, for example), and in comparison to those all other ones are approximately equally fit.

Finally, test series III obviously produced the best results with respect to training as well as validation data (see also Table 17.2). Even more, the results that are given in Table 17.3, column III, and displayed in Figure 17.3, show that the combination of random and roulette parents selection and offspring selection results in a very moderate distribution of the pc_{total} values: Fitter individuals contribute more than less fit ones, but even the worst ones are still able to contribute to a significant extent. Thus, genetic diversity is increased which also contributes positively to the genetic programming process.

17.1.3 Summary

Thus, in order to sum up this section, offspring selection in genetic programming based system identification significantly influences the algorithm's ability to create high quality results as well as the genetic propagation dynamics: Not only fitter individuals are able to propagate their genetic make-up, but also less fit ones are able to contribute to the next population. This is also somehow the case when using proportional selection, but in the presence of individuals with very bad fitness values the selection pressure is almost lost which leads to solutions of rather bad quality. When using offspring selection, extremely bad individuals are eliminated immediately; when using OS in combination with gender specific parent selection (applying random and proportional selection mechanisms), GP is able to produce significantly better results than when using standard techniques. Parents diversification and thus increased genetic diversity in GP populations is considered one of the most influential aspects in this context.

17.1.4 Additional Tests Using Random Parents Selection

In addition to the tests reported on in the previous parts of this section we have also tested conventional as well as extended GP using random parents selection. Thus, we have two more test cases to be analyzed.

As we had expected, the test results obtained for standard GP with random parents selection were very bad; obviously, no suitable models were found. When using OS, on the contrary, the test results for random parents selection were not that bad at all: The models are (on training data) not quite as good as those obtained using random/roulette and OS or conventional GP with tournament parents selection, but still they perform (surprisingly) well on test data[2]. In Table 17.5 we summarize the

[2]Of course, these remarks are only valid for the tests reported on here - we do here not give any general statement regarding result quality using random parents selection and OS.

Table 17.4: Additional GP test strategies.

Strategy	*Properties*
(IV) *Conventional GP*	|Pop| = 2000; Random parents selection nr. of rounds: 500
(V) *Extended GP*	|Pop| = 500; Random parents selection Offspring selection (SuccessRatio = 1, MaxSelPres = 100)

respective result qualities.

In Table 17.6 we give the average pc_{total} values for percentiles of the populations of test series IV and V (collecting the pc_{total} values of 200 indices for forming a partition for series IV and 50 indices for each partition for series V). Obviously (and exactly as we had expected) random parents selection leads to all individuals having the approximately same success in propagating their genetic make-up. When using OS, the result is (even a little bit surprisingly) significantly different: Better individuals have a much higher chance to produce successful offspring than worse ones; the probability of the best 10%, for example, to produce successful children is almost twice as high as the probability of the worst 10% to do so.

Obviously, random parents selection leads to an increased number of generations that have to be executed until a given selection pressure limit is reached. This is graphically shown in Figure 17.4, which gives the selection pressure progress for

Table 17.5: Additional test results (random parents selection).

		IV	*V*
Best Quality (Training)	min.	>50,000.00	5,041.22
	avg.	>50,000.00	7,726.11
	max.	>50,000.00	8,843.73
Best Quality (Test)	min.	>50,000.00	7,129.31
	avg.	>50,000.00	8,412.31
	max.	>50,000.00	12,653.98
Generations		500	102.86
Effort		1,000,000	1,324,302

Table 17.6: Average overall genetic propagation of population partitions for random parents selection tests.

Population Percentile	*Test Strategy*	
	IV	*V*
0	10.03 %	13.16 %
1	10.02 %	12.07 %
2	9.98 %	11.41 %
3	9.99 %	10.66 %
4	10.02 %	10.30 %
5	9.99 %	9.33 %
6	10.00 %	8.96 %
7	9.98 %	8.62 %
8	9.99 %	7.81 %
9	10.00 %	7.68 %

two exemplary test runs of the test series including OS, i.e. III and V. In the standard case using random / roulette parents selection and offspring selection, III, the selection pressure obviously rises faster than when using random parents selection in combination with strict offspring selection. Still, even though it takes longer when using random parents selection, the characteristics are very similar, i.e. it rises steadily with some notable fluctuations.

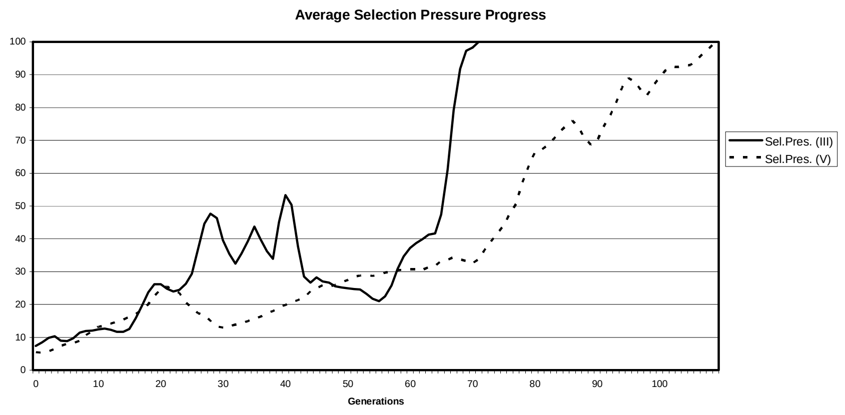


Figure 17.4: Selection pressure progress in two exemplary runs of test series III and V (extended GP with gender specific parents selection and strict offspring selection).

17.2 Variables Diversity

The functions designed for estimating the variables diversity among GP populations described in Section 12.2 have been used for demonstrating internal GP dynamics in standard as well as in extended genetic programming. Thus, again the *Thyroid* and the NO_x data sets, described previously in Chapter 15 and Sections 14.2.1 and 14.2.2, respectively, have been used as basic problems.

In Section 17.2.1 the reader can find a summary of first variables diversity and impact test results previously published in [WAW07c], Section 17.2.2 summarizes systematic variables impact tests using standard and extended GP strategies.

17.2.1 First Exemplary Results

Initial variables diversity and impact tests were, as reported on in [WAW07c], done using the *Thyroid* data set as well as the NO_x data set II (as described in Section 14.2.1); 80% of both data sets were used as identification, 10% as validation and 10% as test data. All tests were executed using 12% mutation rate and single point crossover and mutation operators. The most relevant test settings are summarized in Table 17.7; extended GP hereby stands for GP with gender specific parents selection (random and proportional selection) and strict offspring selection, MSP for *maximum selection pressure.*

Table 17.7: Summary of data and algorithm specific settings of the GP tests investigated in initial variables diversity tests.

Test	*Data Set*	*GP Algorithm*	*Pop. Size*	*Parameter Settings*
I	*Thyroid*	Standard GP	1,000	Number of generations = 1,500 Linear Rank Selection
II	NO_x	Extended GP (with OS)	500	MSP = 200, Success Ratio = 1.0 Random & Roulette Parents Selection
III	NO_x	Extended GP (with OS)	500	MSP = 200, Success Ratio = 0.9 Random & Roulette Parents Selection
IV	*Thyroid*	OS-driven Sliding Window GP	1,000	Sliding Window Moving MSP = 20 MSP = 200, Success Ratio = 1.0 Random & Roulette Parents Selection

Test case IV was taken from the test series executed in the course of investigations of sliding window behavior for GP (see Section 16.2 for details). The main difference here is that the algorithm starts considering only a part of the training data available; after reaching a certain maximum selection pressure, the data scope used for evaluating models is shifted until the end of the data set is reached. This approach significantly increases the speed of GP based structure identification as well as it helps the method to decrease the effects of overfitting. The test case analyzed here is part of test series 5 explained in detail in Section 16.2.

First we report on the results obtained using standard GP: In Figures 17.5, 17.6, and 17.7 selected analysis results for test run I are illustrated. Figure 17.5 shows the impact of all variables over time using the "mean" replacement strategy (defined in Equation 12.10) and the sum of squared differences function for calculating the impact values. Figures 17.6 and 17.7 show the total number of occurrences for all variables at generations 1400 and 1450, respectively. There is obviously no notable variables selection process and also no clear statement possible regarding which variables are more important for modeling the given target variable than others.

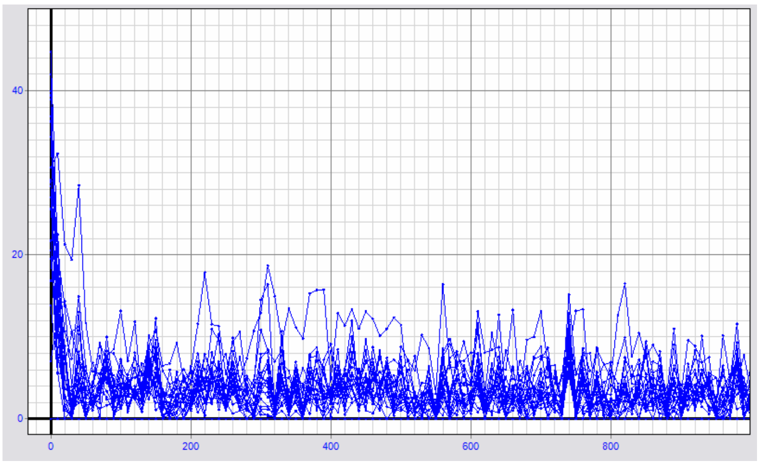


Figure 17.5: Test run I: The impact of all variables is shown over time for the first 1000 generations.

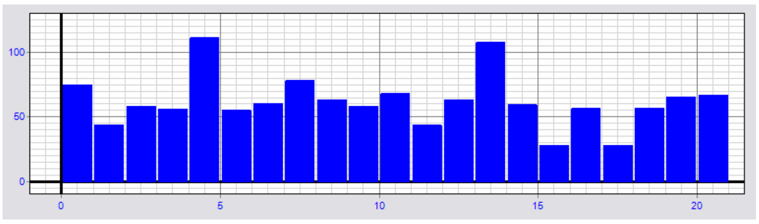


Figure 17.6: Test run I: Total occurrences at generation 1400.

In Figures 17.8, 17.9 and 17.10 we visualize the impact of all variables for test run II. In Figure 17.8 we show the impact using the "linear regression" (see Equations 12.11 – 12.15) replacement strategy and the correlation coefficient (defined in

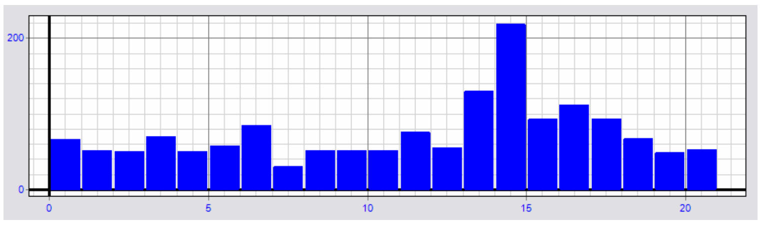


Figure 17.7: Test run I: Total occurrences at generation 1450.

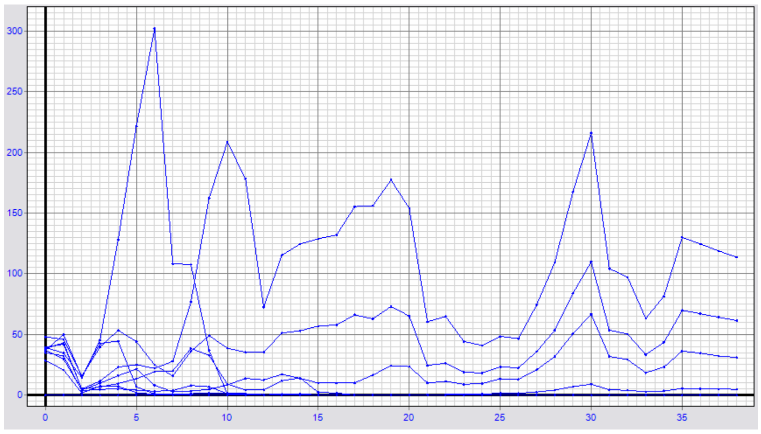


Figure 17.8: Test run II: The impact of all variables over time.

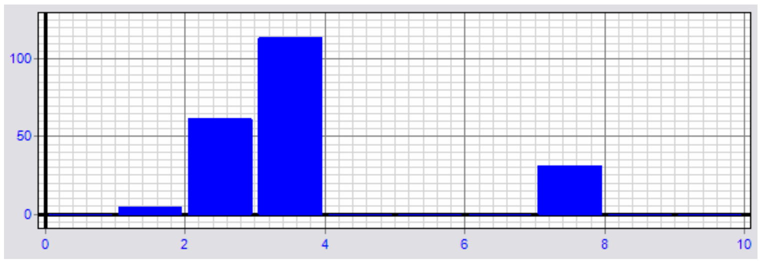


Figure 17.9: Test run II: Final impact analysis, calculated using the r_{mean} / $impact_{msd}$ strategy.

Equation 12.22) impact function; Figures 17.9 and 17.10 show the variables' impact at the end of the algorithm's execution (based on r_{mean} / $impact_{msd}$ and r_{linreg} / $impact_{cc}$ strategies, respectively). Here the variables selection functionality of GP becomes obvious, still the results differ quantitatively depending on the selection of replacement and impact strategies applied.

The Figures 17.11 and 17.12 characterize the algorithmic behavior in test run III: Even though several variables occur rather often in the population, only one variable dominates all other ones with respect to occurrence (Figure 17.11) as well

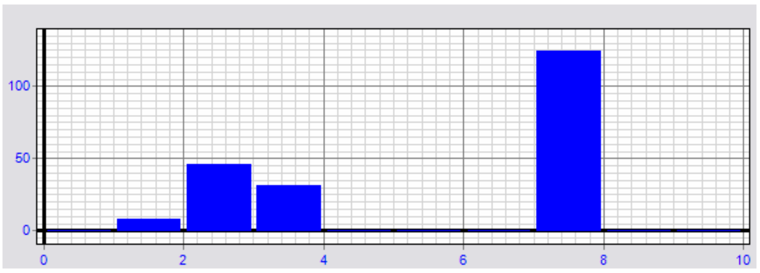


Figure 17.10: Test run II: Final impact analysis, calculated using the r_{linreg} / $impact_{cc}$ strategy.

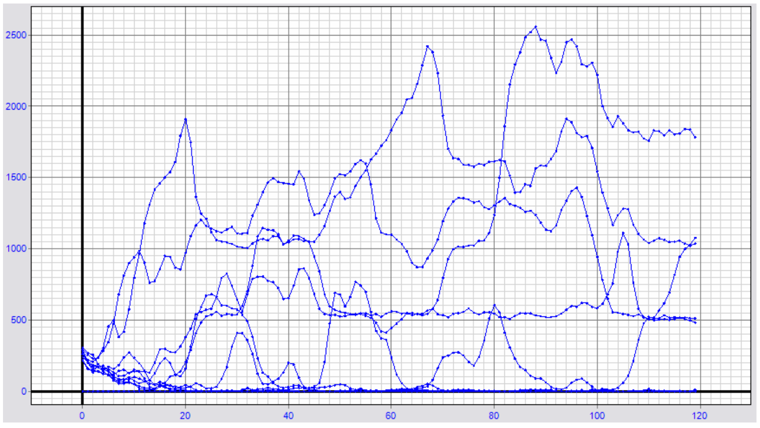


Figure 17.11: Test run III: Occurrence of variables over time.

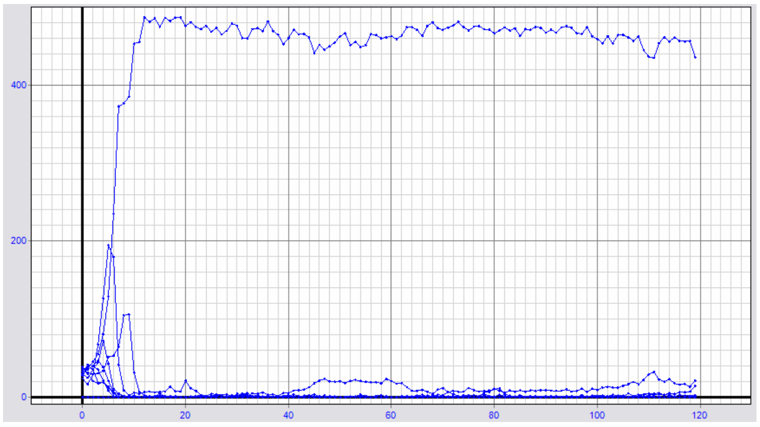


Figure 17.12: Test run III: Impact of variables over time.

as impact (applying r_{linreg} and $impact_{cc}$ strategies, Figure 17.12).

Figures 17.13 and 17.14 finally show the total occurrences of all variables during

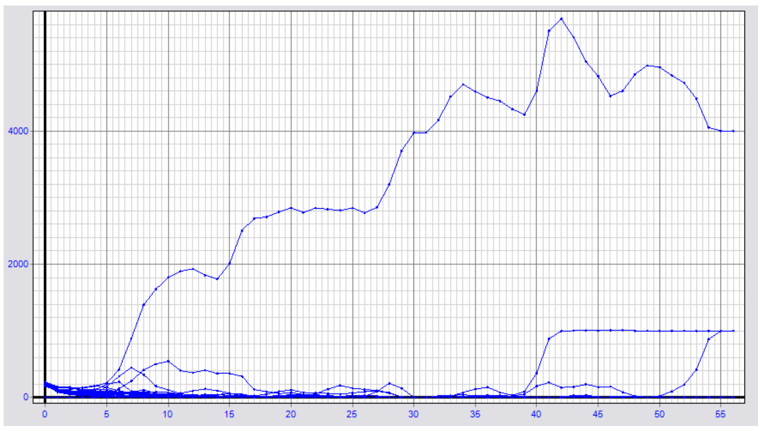


Figure 17.13: Test run IV: Occurrences of variables over time.

the execution of test run IV and the impact of variables (again applying the r_{mean} / $impact_{msd}$ strategy). Here it is even more obvious that one variable dominates all other ones; genetic diversity almost seems to have disappeared since all models only use one variable and simply seem to neglect all other ones.

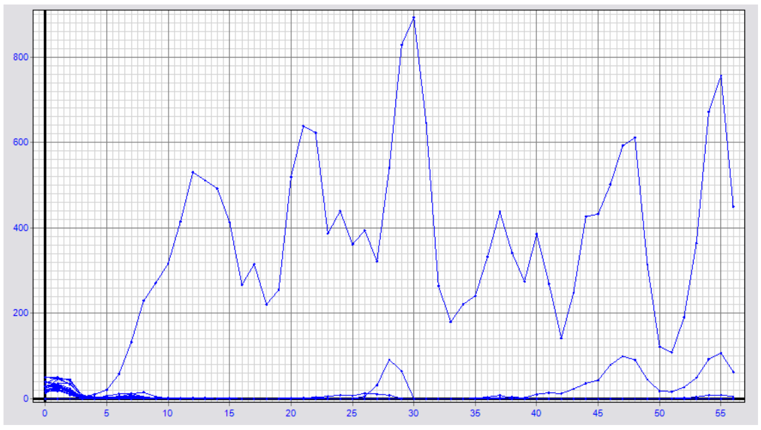


Figure 17.14: Test run IV: Impact of variables over time.

17.2.2 Detailed Analysis, Comparing Standard GP to Extended GP

17.2.2.1 Test Setup

Analyzing the variables diversity and impact test results reported on in the previous section it became clear that a more systematic comparison of standard and extended GP was necessary. So, again using the *Thyroid* data set and the NO_x data set III (see Section 14.2.2 for details), we defined the two test strategies summarized in Table 17.8; we intentionally applied no local adaptation as we wanted to concentrate on similarities or differences of these two GP strategies with respect to variables diversity or impact distribution.

Table 17.8: Summary of data and algorithm specific settings of the systematic GP tests designed for comparing standard to extended GP.

Test	*Data Set*	*GP Algorithm*	*Pop. Size*	*Parameter Settings*
I	NO_x	Extended GP	1,000	MSP = 200, Success Ratio = 1.0 Random & Roulette Parents Selection
II	*Thyroid*	Extended GP	1,000	MSP = 200, Success Ratio = 1.0 Random & Roulette Parents Selection
III	NO_x	Standard GP	1,000	Number of generations = 2,000 Tournament Parents Selection ($k = 3$)
IV	*Thyroid*	Standard GP	1,000	Number of generations = 2,000 Tournament Parents Selection ($k = 3$)

17.2.2.2 Variables Diversity Results

All test cases were executed 5 times independently; the maximum height of the formulae created was set to 6, the maximum structure tree size to 60. For each variable available in the two data sets used we monitored the occurrences (i.e., the number of solutions referencing it, weighted using fitness dependent weighting factors as described in Equations 12.26 and 12.28), for the NO_x tests also their respective impacts (using the r_{mean} / $impact_{msd}$ strategy). As it is not easy to illustrate all these results graphically (at least in a relatively compact way) we see the results summarized in Tables 17.9 – 17.14; for each variable we see the occurrences or impact values, respectively, given as mean values and standard deviations (over the test runs executed) for several selected generations.

Table 17.9: Average weighted numbers of solutions referencing input variables in test runs of GP strategy I.

	Generation 10		*Generation 40*		*Generation 80*		*End of Run*	
	Mean	*StdDev*	*Mean*	*StdDev*	*Mean*	*StdDev*	*Mean*	*StdDev*
qMI	38.1	18.7	385.1	303.7	435.9	182.5	624.0	342.8
pMI	93.9	81.3	34.0	56.1	583.0	719.9	433.0	541.3
HFM	472.4	138.7	952.2	736.7	913.9	553.8	973.6	276.2
N	263.6	76.0	477.5	288.0	589.0	572.0	486.5	251.7
qPI	50.4	20.2	23.8	32.8	0.1	0.1	0.0	0.0
$tiPI$	70.1	63.6	8.4	14.7	0.4	0.8	0.0	0.0
$pRAIL$	50.9	47.8	210.0	181.9	171.2	159.5	224.1	253.4
$pBOOST$	199.4	119.8	55.4	93.3	106.4	184.4	265.3	459.5

Table 17.10: Average weighted numbers of solutions referencing input variables in test runs of GP strategy II.

	Generation 10		*Generation 20*		*Generation 40*		*End of Run*	
	Mean	*StdDev*	*Mean*	*StdDev*	*Mean*	*StdDev*	*Mean*	*StdDev*
$Var00$	5.8	3.7	25.5	35.7	110.6	129.3	185.0	216.8
$Var01$	114.3	128.4	160.7	318.7	152.8	305.0	256.0	511.0
$Var02$	37.2	62.1	68.7	129.8	119.8	141.6	145.6	172.0
$Var03$	4.2	4.1	16.2	20.6	1.1	1.3	1.5	2.5
$Var04$	11.4	1.5	40.2	73.6	0.2	0.2	20.5	40.7
$Var05$	20.2	15.2	2.76	2.9	0.7	0.9	1.4	1.6
$Var06$	11.3	2.2	1.3	1.4	7.3	8.2	26.2	38.0
$Var07$	26.1	27.1	9.6	11.5	88.8	98.8	145.4	169.4
$Var08$	45.6	55.9	62.1	118.4	3.6	2.2	3.5	4.3
$Var09$	78.8	85.3	131.3	225.5	54.1	90.2	90.4	151.3
$Var10$	51.2	91.4	33.3	54.39	111.6	212.4	184.8	357.4
$Var11$	13.9	5.7	1.7	1.41	43.4	77.6	72.1	130.6
$Var12$	79.4	81.9	78.5	127.8	60.2	73.4	62.4	55.3
$Var13$	45.0	33.6	4.4	5.1	15.7	28.8	26.3	48.4
$Var14$	25.1	21.7	1.5	1.2	0.1	0.1	0.2	0.2
$Var15$	36.8	38.0	23.7	34.5	91.0	105.8	136.0	157.1
$Var16$	316.9	397.9	233.5	303.4	441.4	360.6	553.5	616.2
$Var17$	81.8	95.1	93.2	162.6	204.3	284.4	346.3	482.2
$Var18$	63.1	62.4	120.1	209.2	152.3	201.6	28.8	46.5
$Var19$	3.0	1.9	17.2	32.5	0.1	0.08	0.1	0.1
$Var20$	120.4	186.9	237.3	263.0	286.1	369.3	586.7	666.5

Table 17.11: Average weighted numbers of solutions referencing input variables in test runs of GP strategy III.

	Generation 100		*Generation 500*		*Generation 1000*		*Generation 2000*	
	Mean	*StdDev*	*Mean*	*StdDev*	*Mean*	*StdDev*	*Mean*	*StdDev*
qMI	341.4	218.9	329.3	103.6	303.5	102.3	480.2	129.6
pMI	15.6	15.6	53.5	83.5	705.0	1185.6	85.3	142.1
HFM	617.0	298.0	784.6	589.0	1204.9	803.7	1418.4	1005.7
N	426.5	137.3	588.9	270.4	745.3	297.7	945.9	368.0
qPI	7.5	3.2	76.8	126.6	111.0	204.2	143.4	260.2
$tiPI$	81.7	130.6	7.2	1.7	9.9	3.6	17.6	13.7
$pRAIL$	9.4	6.7	187.2	137.5	287.6	219.8	355.9	246.4
$pBOOST$	11.9	9.3	59.3	101.7	82.7	143.7	109.8	182.3

Table 17.12: Average weighted numbers of solutions referencing input variables in test runs of GP strategy IV.

	Generation 100		*Generation 500*		*Generation 1000*		*Generation 2000*	
	Mean	*StdDev*	*Mean*	*StdDev*	*Mean*	*StdDev*	*Mean*	*StdDev*
$Var00$	20.8	34.4	19.5	37.5	31.1	57.1	42.5	81.5
$Var01$	103.8	124.3	141.2	220.8	143.3	208.8	184.9	288.3
$Var02$	2.1	2.1	1.2	0.7	3.4	3.1	44.6	83.1
$Var03$	1.1	0.6	0.6	0.5	2.1	1.1	2.4	1.6
$Var04$	2.4	2.4	0.9	0.3	60.8	117.2	1.7	1.1
$Var05$	18.5	17.5	2.3	1.8	2.2	1.5	2.9	2.4
$Var06$	94.9	112.1	66.5	95.7	54.5	105.0	145.9	170.8
$Var07$	21.5	40.6	57.6	64.8	121.2	86.0	145.4	164.4
$Var08$	16.2	22.9	71.9	49.9	109.9	75.9	149.3	165.9
$Var09$	5.0	8.3	1.3	0.9	5.3	6.3	31.8	57.6
$Var10$	63.8	125.7	123.9	185.4	203.7	315.6	136.8	157.4
$Var11$	3.6	4.0	52.6	102.0	2.4	0.8	3.0	1.9
$Var12$	87.8	173.3	121.8	231.7	172.0	274.8	348.1	598.4
$Var13$	14.2	25.1	33.6	43.6	151.6	175.6	267.6	310.8
$Var14$	23.5	31.1	7.2	11.2	2.1	1.2	4.5	0.7
$Var15$	54.5	107.3	65.0	91.1	95.2	176.1	152.1	15.3
$Var16$	140.0	188.9	266.6	236.3	436.8	370.7	298.2	271.7
$Var17$	2.5	2.1	100.2	196.5	200.7	397.1	186.8	367.1
$Var18$	133.0	264.4	113.7	92.0	220.3	221.1	328.8	389.5
$Var19$	2.8	1.1	40.5	77.1	70.2	136.0	102.0	116.2
$Var20$	56.2	109.8	74.7	96.5	86.7	109.4	90.7	99.9

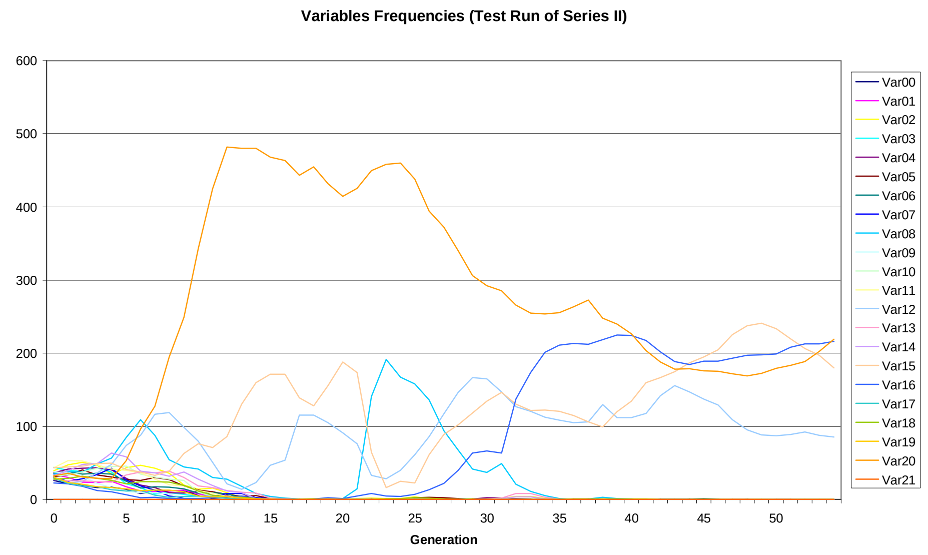


Figure 17.15: Variables diversity over time in an exemplary test run of series II.

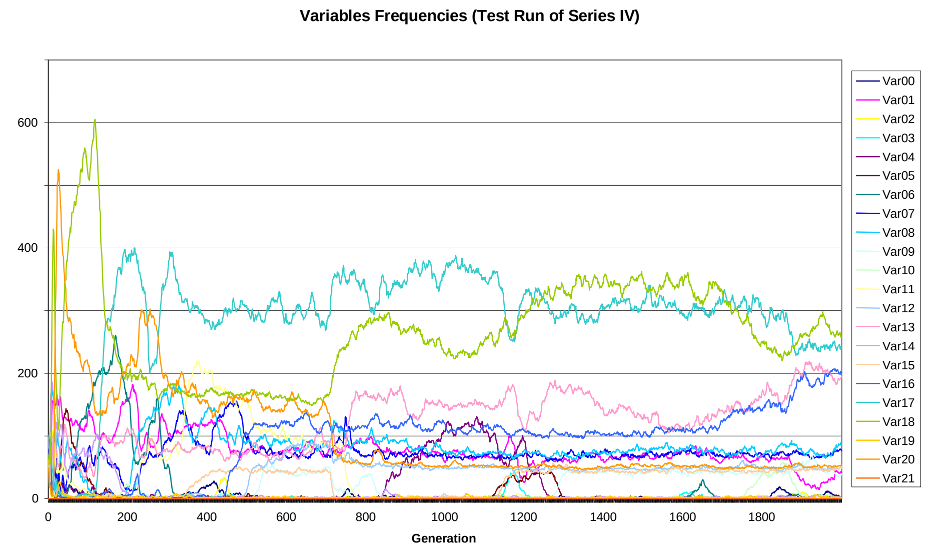


Figure 17.16: Variables diversity over time in an exemplary test run of series IV.

In Figure 17.15 we see the weighted frequency progresses of the variables in an exemplary run of test strategy II, Figure 17.16 visualizes the frequencies progresses in an exemplary run of strategy IV.

Analyzing the statistic features regarding variables frequencies and impacts as well as the Figures 17.15 and 17.16, the following speculation arises: The fluctuation of the calculated frequencies and impact values seem to differ significantly when comparing standard to extended GP; extended GP seems to have a stronger tendency to select variables strictly, i.e. to remove variables from the population completely. Additionally, at least when analyzing the variables impact results for the NO_x tests, the results of the standard GP tests vary more than those produced by extended GP (which can be seen by comparing the results' standard deviations); extended GP here seems to be more stable with respect to variables impact diversity than standard GP. Still, this does not seem to be the case for variables occurrences: The variables' frequencies results of extended GP strategies differ not less, but rather more than those of standard GP tests.

Table 17.13: Average impacts of input variables in test runs of GP strategy I.

	Generation 10		*Generation 40*		*Generation 80*		*End of Run*	
	Mean	*StdDev*	*Mean*	*StdDev*	*Mean*	*StdDev*	*Mean*	*StdDev*
qMI	0.0012	0.0001	0.1233	0.1776	0.1219	0.0893	0.2360	0.2580
pMI	0.0010	0.0019	0.0000	0.0000	0.0167	0.0217	0.0110	0.0139
HFM	0.0640	0.0256	0.0981	0.0643	0.1525	0.0876	0.2873	0.2673
N	0.0029	0.0008	0.0251	0.0245	0.1039	0.0706	0.0987	0.0984
qPI	0.0004	0.0007	0.0006	0.0012	0.0136	0.0271	0.0000	0.0000
tiPI	0.0027	0.0053	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
pRAIL	0.0011	0.0019	0.0107	0.0125	0.0257	0.0303	0.0181	0.0214
pBOOST	0.0000	0.0000	0.0000	0.0000	0.0001	0.0002	0.0000	0.0001

Table 17.14: Average impacts of input variables in test runs of GP strategy III.

	Generation 100		*Generation 500*		*Generation 1000*		*Generation 2000*	
	Mean	*StdDev*	*Mean*	*StdDev*	*Mean*	*StdDev*	*Mean*	*StdDev*
qMI	0.1471	0.1351	0.0374	0.0399	0.0904	0.1539	0.2599	0.4494
pMI	0.0225	0.0380	0.1262	0.1584	0.1138	0.0964	0.1189	0.1952
HFM	0.1366	0.0797	0.0809	0.0701	0.0008	0.0015	0.1515	0.1148
N	0.0887	0.1527	0.0539	0.0703	0.0599	0.0807	0.0844	0.0564
qPI	0.1700	0.1586	0.1314	0.0488	0.2656	0.1074	0.1207	0.0766
tiPI	0.0813	0.1263	0.0591	0.0123	0.0776	0.1241	0.1339	0.1857
pRAIL	0.0854	0.1274	0.0681	0.1068	0.0519	0.0899	0.0802	0.0722
pBOOST	0.0238	0.0413	0.0153	0.0266	0.0833	0.0732	0.3104	0.1828

17.2.2.3 Variables Diversity Fluctuation Analysis

In order to compare standard GP to extended GP a bit more systematically with respect to fluctuations in variables diversity we calculate the average differential values of the variables frequencies or impact values, respectively, and then sum up all these values:

$$x'_t = x_{t-1} - x_t \tag{17.2}$$

$$x'_{total} = \frac{1}{n-1} \sum_{i \in [2;n]} |x'_i| \tag{17.3}$$

where $x_1 \ldots x_n$ denotes the frequency or impact values monitored over n generations.

Of course, this way of comparing GP incorporating OS to standard GP is not quite fair because the numbers of generations differ a lot (the number of generations in standard GP is here set to 2,000 whereas extended GP is here finished after 100 to 150 rounds). The number of individuals created and evaluated per generation is fixed in standard GP (i.e., equal to the population's size), in GP with offspring selection this number can be a lot higher (depending on OS parameters, for example it can become 10 or 50 times as much as the population's size). Thus, one could bring forward the argument that it would be fairer to calculate the fluctuation in standard GP as the differential of order 10 instead of 1:

$$x'^{10}_t = x_{t-10} - x_t \tag{17.4}$$

$$x'_{total_{10}} = \frac{1}{n-10} \sum_{i \in [11;n]} |x'^{10}_t| \tag{17.5}$$

In the Tables 17.15 and 17.16 we summarize the x'_{total} values for the variable's frequencies and impact values, given as mean average values; for the standard GP tests the $x'_{total_{10}}$ values are also stated.

The results given in Table 17.15, which summarizes the fluctuations in NO_x tests, show that variables frequencies fluctuate (with respect to first order derivative) more in OS GP than in standard GP, but when looking also at the $x'_{total_{10}}$ values we see that the extended GP test runs seem to have run more smoothly (with respect to variables frequencies) than standard GP runs.
The results for the *Thyroid* tests are even more univocal: The GP runs incorporating roulette and random parents selection as well as offspring selection partially show less variables frequencies fluctuation than the standard GP tests, even considering the first order derivatives; again, looking at the $x'_{total_{10}}$ values we see that extended GP seems to have run a lot more smoothly than standard GP.

Table 17.15: Fluctuation of variable frequencies for test strategies I and III.

Variable Name	x'_{total}(frequencies) Strategy I	Strategy III	$x'_{total_{10}}$(frequencies) Strategy III
qMI	18587.44	524.32	5296.70
pMI	158.35	124.28	4789.46
HFM	17675.91	973.60	7908.22
N	14127.64	1164.80	7132.37
qPI	569.81	387.16	3185.82
$tiPI$	332.09	27.15	403.80
$pRAIL$	297.62	154.97	2452.97
$pBOOST$	1248.82	166.95	5373.77

Table 17.16: Fluctuation of variable frequencies for test strategies II and IV.

Variable Name	x'_{total}(frequencies) Strategy II	Strategy IV	$x'_{total_{10}}$(frequencies) Strategy IV
$Var00$	56.82	115.38	144.06
$Var01$	92.25	1138.44	2327.61
$Var01$	245.13	1865.43	6425.16
$Var03$	187.83	132.33	242.7
$Var04$	170.1	103.89	264.48
$Var05$	464.4	172.8	1001.76
$Var06$	120.6	182.76	658.89
$Var07$	118.5	639.69	1901.4
$Var08$	523.92	1124.73	2375.37
$Var09$	122.91	130.92	620.55
$Var10$	140.67	302.82	334.38
$Var11$	238.35	147.24	2201.91
$Var12$	315.06	726.78	419.94
$Var13$	153.51	348.51	3539.82
$Var14$	245.25	220.23	553.44
$Var15$	552.66	268.08	413.22
$Var16$	805.23	297.42	1295.91
$Var17$	74.04	711	1082.22
$Var18$	98.34	5792.28	11128.83
$Var19$	83.07	68.19	52.98
$Var20$	2658.24	2327.19	9436.5

17.3 Single Population Diversity Analysis

17.3.1 GP Test Strategies

Within our first series of empirical tests regarding solutions similarity and diversity we analyzed the diversity of populations of single population GP processes. For testing the population diversity analysis method described in Section 12.4 and illustrating graphical representations of the results of these tests we have used the following two data sets:

- The NO_x data set contains the measurements taken from a 2 liter 4 cylinder BMW diesel engine at a dynamical test bench (simulated vehicle: BMW 320d Sedan); this data set has already been described in Section 14.2.2.
- The *Thyroid* data set is a widely used machine learning benchmark data set containing the results of medical measurements which were recorded while investigating patients potentially suffering from hypothyroidism; further details regarding this data set can be found in Chapter 15.

Both data collections have been split into training and validation / test data partitions taking the first 80% of each data set as training samples available to the identification algorithm; the rest of the data is considered as validation data.

We have used various GP selection strategies for analyzing the NO_x and the *Thyroid* data sets:

- On the one hand, we have used standard GP with proportional as well as tournament selection (tournament size $k = 3$).
- On the other hand we have also intensively tested GP using offspring selection and gender specific parents selection (proportional and random selection).

In general, we have tested GP with populations of 1,000 solution candidates (with a maximum tree size of 50 and a maximum tree height of 5), standard subtree exchange crossover, structural as well as parametric node mutation and total 15% mutation rate; the mean squared errors function was used for evaluating the solutions on training as well as on validation (test) data. Other essential parameters vary depending on the test strategies; these are summarized in Table 17.17.

Table 17.17: GP test strategies.

Strategy	**Properties**
(A) *Standard GP*	Tournament parents selection (tournament size $k = 3$); Number of generations: 4000
(B) *Standard GP*	Proportional parents selection; Number of generations: 4000
(C) *GP with OS*	Gender specific parents selection; (Random & proportional) Success ratio: 0.8 Comparison factor: 0.8 (Maximum selection pressure: 50 (not reached) Number of generations: 4000
(D) *GP with OS*	Gender specific parents selection; (Random & proportional) Success ratio: 1.0 Comparison factor: 1.0 Maximum selection pressure: 100

17.3.2 Test Results

In Table 17.18 we summarize the quality of the best models produced using the GP test strategies (A) – (D); for the NO_x data set the quality is given as the mean squared error, for the *Thyroid* data set we give the classification accuracy, i.e. the ratio of samples that are classified correctly. The models are evaluated on training as well as on validation data; as each test strategy was executed 5 times independently, we here state mean average and standard deviation values.
Obviously, the test series (A) and (D) perform best; the results produced using offspring selection are better than those using standard GP. The classification results for the *Thyroid* data set are not quite as good as those reported in [WAW06e] and Section 8.2; this is due to the fact that we here used smaller models and concentrated on the comparison of GP strategies with respect to population diversity.

Solution quality analysis is of course important and interesting, but here we are more interested in a comparison of population diversity during the execution of the GP processes. We have calculated the similarity among the GP populations during

Table 17.18: Test results: Solution qualities.

Results for NO_x test series

	GP Strategy (A)	**(B)**	**(C)**	**(D)**
Training (*mse*)	2.518	5.027	2.674	1.923
Training (*std*(*mse*))	1.283	2.142	2.412	0.912
Validation (*mse*)	3.012	5.021	2.924	2.124
Validation (*std*(*mse*))	1.431	3.439	2.103	1.042
Evaluated solutions, avg.	$4 \cdot 10^6$	$4 \cdot 10^6$	$10.2 \cdot 10^6$	$3.91 \cdot 10^6$
Generations (avg.)	4,000	4,000	4,000	98.2

Results for *Thyroid* test series

	GP Strategy (A)	**(B)**	**(C)**	**(D)**
Training (cl. acc., avg.)	0.9794	0.9758	0.9781	0.9812
Training (cl. acc., *std*)	0.0032	0.0017	0.0035	0.0012
Validation (cl. acc., avg.)	0.9764	0.9675	0.9767	0.9804
Validation (cl. acc., *std*)	0.0029	0.0064	0.0069	0.0013
Evaluated solutions, avg.	$4 \cdot 10^6$	$4 \cdot 10^6$	$12.2 \cdot 10^6$	$5.1 \cdot 10^6$
Generations (avg.)	4,000	4,000	4,000	167.8

the execution of the GP test series described in Table 17.17: The multiplicative similarity approach (as defined in Equations 11.21 – 11.23) has been chosen; all coefficients $c_1 \dots c_{10}$ were set to 0.2, only the coefficient c_1 weighting the level difference contribution d_1 was set to 0.8.

In Table 17.19 we give the average population similarity values calculated using Equation 12.51; again, as each test series was executed several times, we give the average and standard deviation values (written in italic letters). As we see in the first row, the average similarity values are approximately in the interval [0.2; 0.25] at the beginning of the GP runs, i.e. after the initialization of the GP populations. In standard GP, as can be seen in the first column, the average similarity reaches values above 0.7 after 400 generations and stays at approximately this level until the end of the execution of the GP process; in the end, the average similarity was ∼0.87 in the NO_x tests and ∼0.81 in the *Thyroid* test series. Analyzing the second and the third column we notice that this is not the case in test series (B) and (C): The similarity values do in test series (B) by far not rise as high as in series (A) (especially when working on the *Thyroid* data set), and also in test series (C) we have measured significantly lower similarities than in series (A) (i.e., the population diversity was higher during the whole GP process). Obviously, the use of offspring selection with rather soft parameter settings (i.e., success ratio and comparison factor set to values below 1.0) does not have the same effects on the GP process as strict ones. The by far highest similarity values are documented for test series (D) using maximally strict offspring selection (which has produced the best quality models, as documented in Table 17.18): As is summarized in the far right column, during the whole evolutionary process the mutual similarity among the models increases steadily, while also the selection pressure increases. In the end, when the selection pressure reaches a high level (in these cases, the predefined limit was set to 100) and the algorithm stops, we see a very high similarity among the solution candidates, i.e. the population has converged and evolution is likely to have gotten stuck. This is in fact consistent with the impression already stated in [WAW06a] or [WAW06e], e.g.; here we see that this in fact really happens.

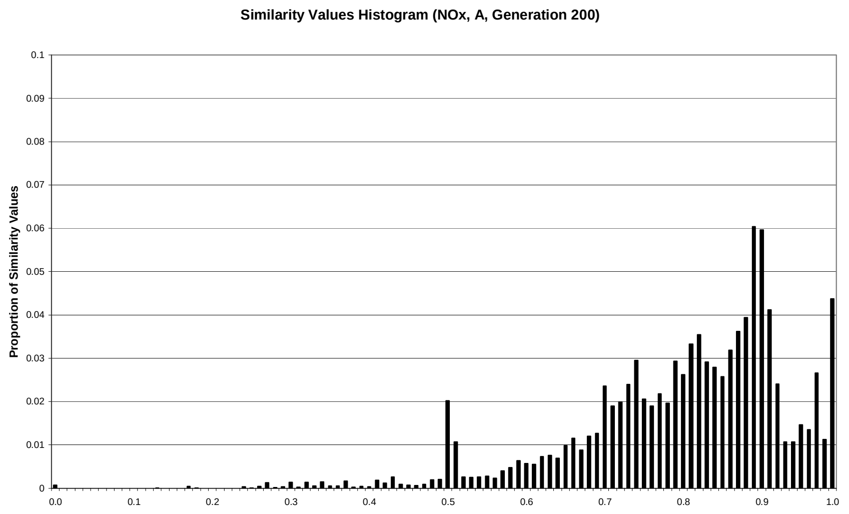


Figure 17.17: Distribution of similarity values in an exemplary run of NO_x test series A, generation 200.

In Table 17.20 we summarize the maximum population diversity values calculated using Equation 12.52; again we give the average and standard deviation values (written in italic letters). As we see in the first (left) column, in standard GP with tournament selection the average maximum similarity reaches values above 0.95 rather fast, i.e. for all models in the population rather similar solutions can be found. This is not the case when using proportional selection. When using offspring selection the same effect as in standard GP with tournament selection can be seen, especially in the NO_x test series.

The Figures 17.17 – 17.20 exemplarily show the average population diversity by

Table 17.19: Test results: Population diversity (average similarity values; avg, std).

***NO_x* tests**					
Gen.	**GP Strategy**			**Gen.**	**GP Strategy**
	(A)	**(B)**	**(C)**		**(D)**
0	0.247	0.250	0.270	0	0.197
	0.041	*0.031*	*0.037*		*0.039*
100	0.723	0.491	0.517	10	0.397
	0.073	*0.051*	*0.038*		*0.039*
400	0.813	0.497	0.564	20	0.603
	0.035	*0.058*	*0.059*		*0.049*
1000	0.859	0.510	0.520	40	0.810
	0.021	*0.055*	*0.052*		*0.039*
4000	0.871	0.518	0.526	End of	0.985
(End of run)	*0.019*	*0.059*	*0.053*	run	*0.032*

***Thyroid* tests**					
Gen.	**GP Strategy**			**Gen.**	**GP Strategy**
	(A)	**(B)**	**(C)**		**(D)**
0	0.206	0.205	0.208	0	0.197
	0.041	*0.040*	*0.036*		*0.040*
100	0.581	0.241	0.444	10	0.397
	0.047	*0.043*	*0.035*		*0.039*
400	0.737	0.321	0,610	20	0.602
	0.032	*0.058*	*0.026*		*0.049*
1000	0.808	0.341	0.692	40	0.810
	0.029	*0.049*	*0.031*		*0.041*
4000	0.812	0.343	0.701	End of	0.975
(End of run)	*0.038*	*0.056*	*0.030*	run	*0.019*

giving the distribution of similarities among all individuals. The Figures 17.17 and 17.18 show the similarity distributions of an exemplary test run of series (A) at generation 200 and 4000; obviously, most similarity calculations returned similarity values between 0.7 and 1.0, and the distribution at generation 200 is comparable to the distribution at the end of the test run. For the GP runs incorporating offspring selection this is not the case, as we exemplarily see in Figures 17.19 and 17.20: After 20 generations most similarity values almost fit Gaussian distribution with mean value 0.8, and at the end of the run all models are very similar to each other (i.e., the population has converged, the selection pressure reaches the given limit

Table 17.20: Test results: Population diversity (maximum similarity values; avg, std).

***NO_x* tests**					
Gen.	**GP Strategy (A)**	**(B)**	**(C)**	**Gen.**	**GP Strategy (D)**
0	0.919 *0.116*	0.934 *0.095*	0.904 *0.123*	0	0.936 *0.109*
100	0.995 *0.014*	0.825 *0.074*	0.944 *0.059*	10	0.961 *0.049*
400	0.998 *0.006*	0.809 *0.075*	0.978 *0.037*	20	0.971 *0.033*
1000	0.999 *0.005*	0.811 *0.059*	0.965 *0.044*	40	0.995 *0.012*
4000 (End of run)	0.999 *0.003*	0.819 *0.066*	0.969 *0.035*	End of run	0.996 *0.009*

***Thyroid* tests**					
Gen.	**GP Strategy (A)**	**(B)**	**(C)**	**Gen.**	**GP Strategy (D)**
0	0.823 *0.127*	0.771 *0.145*	0.766 *0.145*	0	0.777 *0.157*
100	0,958 *0.028*	0.749 *0.123*	0.840 *0.094*	10	0.873 *0.101*
400	0.973 *0.032*	0.752 *0.125*	0.883 *0.067*	20	0.934 *0.049*
1000	0.977 *0.022*	0.744 *0.117*	0.913 *0.061*	40	0.976 *0.022*
4000 (End of run)	0.977 *0.021*	0.754 *0.111*	0.909 *0.058*	End of run	0.999 *0.004*

and the algorithm stops).

Finally, Figure 17.21 shows the average similarity values for each model (calculated using Equation 12.49) for exemplary test runs of the *Thyroid* test series (A)[3] and (D). Obviously, the average similarity in standard GP reaches values in the range [0.7;0.8] very early and then stays at this level during the rest of the GP

[3]In fact, for the test run of series (A) we here only show the progress over the first 2000 generations.

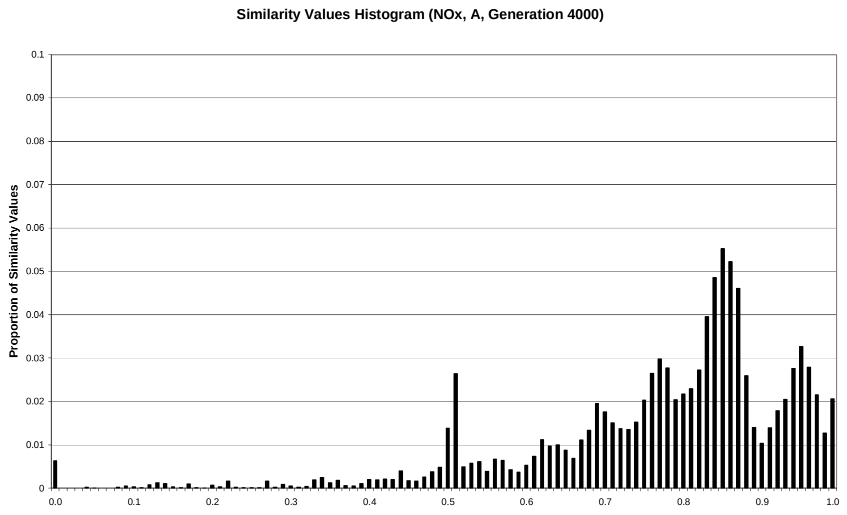


Figure 17.18: Distribution of similarity values in an exemplary run of NO_x test series A, generation 4000.

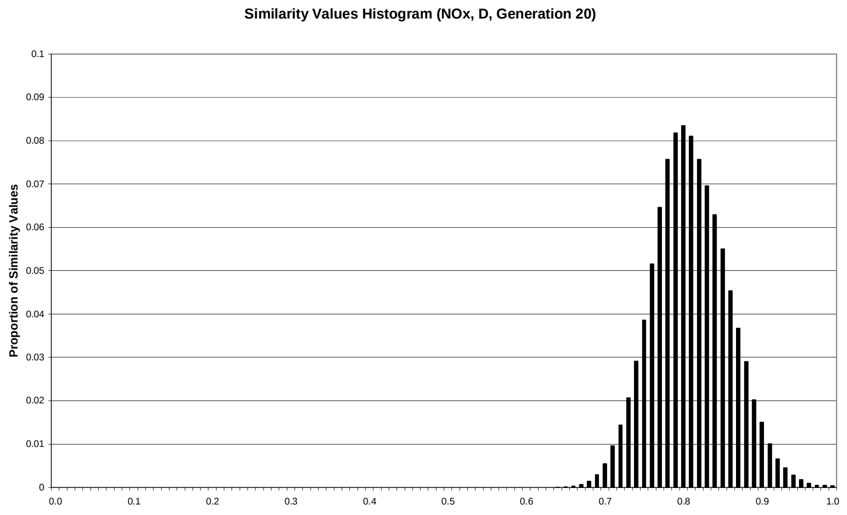


Figure 17.19: Distribution of similarity values in an exemplary run of NO_x test series (D), generation 20.

execution. When using gender specific selection and offspring selection, otherwise, the average similarity steadily increases during the GP process and almost reaches 1.0 at the end of the run, when the maximum selection pressure is reached.

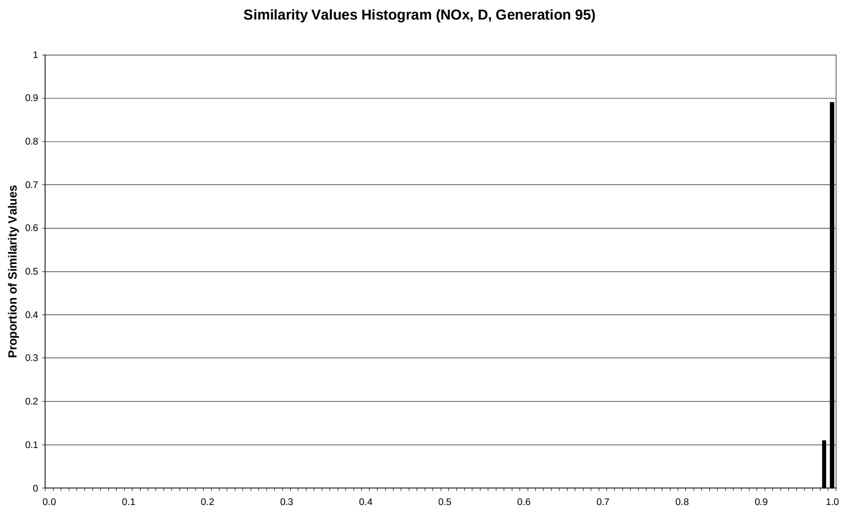


Figure 17.20: Distribution of similarity values in an exemplary run of NO_x test series (D), generation 95.

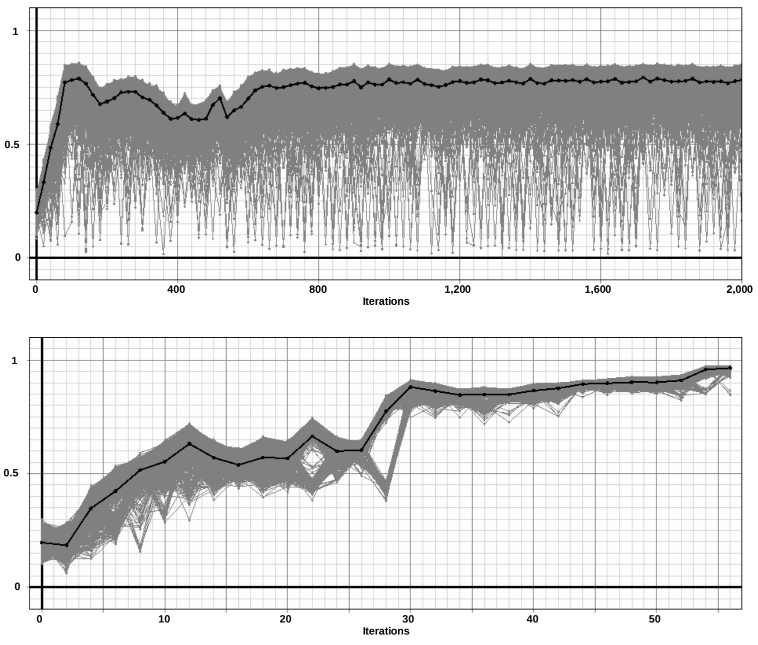


Figure 17.21: Population diversity progress in exemplary *Thyroid* test runs of series (A) and (D) (shown in the upper and lower graph, respectively).

17.3.3 Conclusion

Structural similarity estimation has been used for measuring the genetic diversity among GP populations: Several variations of genetic programming using different types of selection schemata have been tested using fine-grained similarity estimation, and two machine learning data sets have been used for these empirical tests. The test results presented show that population diversity differs a lot in the test runs depending on the selection schemata used.

17.4 Multi Population Diversity Analysis

Our second series of empirical tests regarding solutions similarity and diversity was dedicated to the diversity of populations of multi population GP processes; for testing the multi population diversity analysis method described in Section 12.4 and illustrating graphical representations of the results of these tests we have again used the following two data sets: The NO_x data set described in Section 14.2.2 as well as the *Thyroid* data set.

Both data collections have been split into training and validation / test data partitions; in the case of the NO_x data set the first 50% of the data set were used as training samples, in the case of the *Thyroid* data set the first 80% were considered by the training algorithms.

17.4.1 GP Test Strategies

In general, 4 different strategies for parallel genetic programming have been applied:

- Parallel island GP without interaction between the populations; i.e., all populations evolve independently.
- Parallel island GP with occasional migration after every 100th generation in standard GP and every 5th generation in GP with offspring selection: The worst 1% of each population p_i is replaced by copies of the best 1% of solutions in population p_{i-1}; the best solutions of the last population (in the case of n population that is p_n) replace the worst ones of the first population (p_1). The unidirectional ring migration topology has been used.
- Parallel island GP with migration after every 50th generation in standard GP and every 5th generation in GP with offspring selection: The worst 5% of each population p_i is replaced by copies of the best 5% of solutions in population p_{i-1}. Again, the unidirectional ring migration topology has been used.
- Finally, the SASEGASA algorithm as described in Section 5.1.6 has been used as well.

In all cases the algorithms have been initialized with 5 populations, each containing 200 solutions (in our case representing formulas, of course). Additionally, each of the first 3 strategies has been tested with standard GP settings as well as offspring selection; Table 17.21 summarizes the 7 test strategies that have been applied and whose results shall be discussed here.

Table 17.21: GP test strategies.

Strategy	**Properties**
(A) *Parallel standard GP*	Tournament parents selection (tournament size $k = 3$); Number of generations: 2000
(B) *Parallel GP with OS*	Random & roulette parents selection Strict Offspring selection (success ratio: 1.0, (comparison factor: 1.0, maximum selection pressure: 200)
(C) *Parallel standard GP, 1% migration*	Tournament parents selection (tournament size $k = 3$); Number of generations: 2000 1% best / worst replacement after every 100th generation
(D) *Parallel GP with OS, 1% migration*	Random & roulette parents selection Strict Offspring selection (success ratio: 1.0, (comparison factor: 1.0, maximum selection pressure: 200) 1% best / worst replacement after every 5th generation
(E) *Parallel standard GP, 5% migration*	Tournament parents selection (tournament size $k = 3$); Number of generations: 2000 5% best / worst replacement after every 50th generation
(F) *Parallel GP with OS, 5% migration*	Random & roulette parents selection Strict Offspring selection (success ratio: 1.0, (comparison factor: 1.0, maximum selection pressure: 200) 5% best / worst replacement after every 5th generation
(G) *SASEGASA*	Random & roulette parents selection Strict Offspring selection (success ratio: 1.0, (comparison factor: 1.0, maximum selection pressure: 200)

17.4.2 Test Results

All test strategies summarized in Table 17.21 have been executed 5 times using the NO_x as well as the *Thyroid* data set. Multi population diversity was measured using the equations given in Section 12.4.2: For each solution we calculate the average as well as the maximum similarities with solutions of all other populations of the respective algorithms (in the following, these values are denoted as $MPdiv$ values). Additionally, we have also collected all solutions of the algorithms' populations into temporary total populations and calculate the average as well as the maximum similarities of all solutions compared to all other ones (hereafter denoted

as $SPdiv$ values).
Again, the multiplicative structural similarity approach (as defined in Equations 11.21 – 11.23) has been used for estimating the similarity of model structures; all coefficients $c_1 \ldots c_{10}$ were set to 0.2, only the coefficient c_1 weighting the level difference contribution d_1 was set to 0.8. The similarity of models was calculated symmetrically (as described in Equation 12.48).

In the following we summarize these values for all test runs by stating the average values as well as standard deviations: Table 17.22 summarizes the results of the test runs using the *Thyroid* data set, 17.23 those of the test runs using the NO_x data set.

Figure 17.22 exemplarily illustrates the multi population diversity in a test run of series F at iteration 50: The value represented in row i of column j in bar k gives the average similarity of model i of population k with all formulas stored in population j. Low multi population similarity values are indicated by light cells, dark cells represent high similarity values.

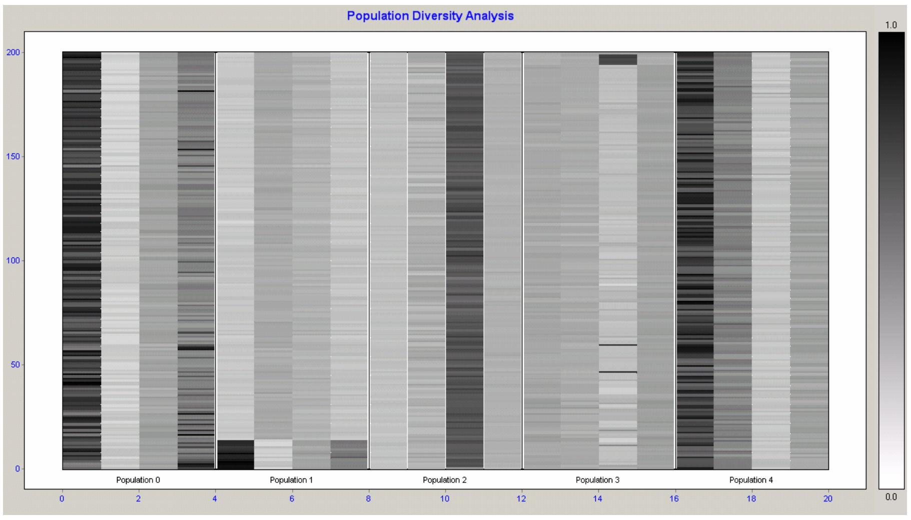


Figure 17.22: Exemplary multi-population diversity of a test run of *Thyroid* series F at iteration 50.

Table 17.22: Multi population diversity test results of the GP test runs using the *Thyroid* data set.

Results for the *Thyroid* data set						
Test Series	**Iteration**		**MPdiv (avg)**	**MPdiv (max)**	**SPdiv (avg)**	**SPdiv (max)**
A	300	avg	0.2433	0.3301	0.2048	0.8973
		std	*0.0514*	*0.0496*	*0.0612*	*0.0291*
	2000	avg	0.3592	0.3925	0.3628	0.9027
		std	*0.0613*	*0.0610*	*0.0593*	*0.0351*
B	20	avg	0.1698	0.2356	0.2130	0.9182
		std	*0.0497*	*0.0317*	*0.0317*	*0.0852*
	End of	avg	0.3915	0.4037	0.3592	0.9850
	Run	*std*	*0.0599*	*0.0769*	*0.0820*	*0.0202*
C	300	avg	0.1778	0.2788	0.1836	0.6543
		std	*0.0587*	*0.0549*	*0.0296*	*0.0971*
	2000	avg	0.4145	0.4885	0.3834	0.9236
		std	*0.0551*	*0.0762*	*0.0665*	*0.0417*
D	20	avg	0.3276	0.4269	0.3394	0.9312
		std	*0.0486*	*0.1094*	*0.0175*	*0.0459*
	End of	avg	0.4412	0.5822	0.3866	0.9736
	Run	*std*	*0.0734*	*0.0635*	*0.0772*	*0.0283*
E	300	avg	0.3395	0.6271	0.2715	0.6116
		std	*0.0441*	*0.0975*	*0.0139*	*0.0811*
	2000	avg	0.5329	0.8710	0.3991	0.9129
		std	*0.0833*	*0.0509*	*0.0921*	*0.0821*
F	20	avg	0.3721	0.5024	0.2711	0.5192
		std	*0.0629*	*0.0822*	*0.0981*	*0.0601*
	End of	avg	0.5915	0.8802	0.4576	0.9828
	Run	*std*	*0.1034*	*0.0996*	*0.0514*	*0.0437*
G	20	avg	0.4839	0.5473	0.3173	0.5237
		std	*0.0823*	*0.0419*	*0.0581*	*0.0623*
	50	avg	0.4325	0.5512	0.3228	0.5828
		std	*0.0518*	*0.0920*	*0.0672*	*0.0660*
	100	avg	0.5102	0.7168	0.3783	0.7296
		std	*0.0730*	*0.0724*	*0.0861*	*0.0740*
	200	avg	0.8762	0.9314	0.4206	0.9512
		std	*0.0505*	*0.0458*	*0.0792*	*0.0249*
	End of	avg	–	–	0.9792	0.9934
	Run	*std*	–	–	*0.0256*	*0.0162*

Table 17.23: Multi population diversity test results of the GP test runs using the NO_x data set.

Results for the NO_x data set						
Test Series	**Iteration**		**MPdiv (avg)**	**MPdiv (max)**	**SPdiv (avg)**	**SPdiv (max)**
A	300	avg	0.3187	0.3991	0.2773	0.8613
		std	*0.0124*	*0.0685*	*0.0726*	*0.0799*
	2000	avg	0.3689	0.4627	0.3300	0.9887
		std	*0.0288*	*0.0390*	*0.0390*	*0.0434*
B	20	avg	0.1997	0.1498	0.2902	0.8992
		std	*0.0698*	*0.0912*	*0.0604*	*0.0634*
	End of	avg	0.3723	0.4811	0.3440	0.9743
	Run	*std*	*0.0233*	*0.0244*	*0.0254*	*0.0482*
C	300	avg	0.2515	0.3323	0.1935	0.8293
		std	*0.0968*	*0.0685*	*0.0607*	*0.1062*
	2000	avg	0.3329	0.4741	0.2821	0.9311
		std	*0.0365*	*0.0323*	*0.0402*	*0.0441*
D	20	avg	0.2985	0.3922	0.3791	0.8862
		std	*0.0870*	*0.0825*	*0.0487*	*0.0829*
	End of	avg	0.5544	0.6839	0.4208	0.9661
	Run	*std*	*0.0542*	*0.1039*	*0.0280*	*0.0332*
E	300	avg	0.5002	0.6697	0.3111	0.6037
		std	*0.0588*	*0.0696*	*0.0474*	*0.0453*
	2000	avg	0.6002	0.8523	0.4745	0.9763
		std	*0.0538*	*0.0263*	*0.0728*	*0.0910*
F	20	avg	0.3597	0.5248	0.3901	0.5839
		std	*0.0743*	*0.0769*	*0.0662*	*0.0775*
	End of	avg	0.5607	0.9080	0.4877	0.9906
	Run	*std*	*0.0931*	*0.0799*	*0.0249*	*0.0181*
G	20	avg	0.4471	0.5303	0.2694	0.4670
		std	*0.0619*	*0.0897*	*0.0802*	*0.0522*
	50	avg	0.4923	0.6102	0.3025	0.6120
		std	*0.0854*	*0.0749*	*0.0550*	*0.0902*
	100	avg	0.5889	0.6939	0.3923	0.7972
		std	*0.1184*	*0.0835*	*0.0812*	*0.0805*
	200	avg	0.9047	0.9148	0.5741	0.9128
		std	*0.0387*	*0.0258*	*0.1253*	*0.0401*
	End of	avg	–	–	0.9683	0.9932
	Run	*std*	–	–	*0.0412*	*0.0319*

17.4.3 Discussion

As we see in Tables 17.22 and 17.23, the average diversity among populations in parallel island GP without interaction (i.e., in test series (A) and (B)) rises up to values between 0.35 and 0.4, no matter whether or not OS is applied; the maximum values eventually reach values between 0.45 and 0.5. Considering all solutions collected in temporary total populations, as expected the average similarities reach values below 0.4, the maximum similarities almost reach 1.0.

The similarity values monitored in test series (C) and (D) are, in comparison to those of series (A) and (B), slightly higher, but not dramatically. This does not hold for the next pair of test series (with 5% migration): The similarity values calculated for test series (E) and (F) are significantly higher than those of test series (A) – (D); in other words, the exchange of only 5% of the populations' models can lead to a significant decrease of population diversity among populations of multi population GP.

When using the SASEGASA, the diversity among populations is high in the beginning and then steadily decreases as the algorithm is executed. This is of course due to the reunification of populations as soon as the maximum selection pressure is reached.

By executing these test series and analyzing the results as given in this section we have demonstrated how multi population diversity can be monitored using similarity measures as those described in Chapter 11. Reference values are given by parallel GP without migration; of course, the higher the migration rates become, the more migration affects the diversity among GP populations. When using the SASEGASA, rather high multi population specific diversity is given in the early stages of the parallel GP process, and due to the merging of population the diversity decreases and in the end reaches diversity values comparable to those of single population GP with offspring selection.

17.5 Comparison of Population Diversity Measures

Obviously, structural similarity estimation of formula trees is not equivalent to evaluation based similarity estimation; still, somehow we expect a significant correlation between the results calculated using these two approaches. In order to get an overview regarding this issue, we have analyzed a series of GP tests including both similarity estimation strategies.

17.5.1 Test Setup

In detail, we have here chosen the GP test strategies which are summarized in Table 17.24:

Table 17.24: GP test strategies for similarity estimators comparison.

GP Strategy	*Parameters*
Standard GP (SGP)	Population size: 1000, mutation rate: 15% Parents selection: Tournament selection ($k = 3$) Number of generations: 2000
Extended GP (EGP)	Population size: 1000, mutation rate: 15% Parents selection: Gender specific selection (random & roulette) Offspring selection (success ratio: 1, comparison factor: 1, maximum selection pressure: 100)

The following similarity estimation functions are used:

- Evaluation based similarity estimation: As described in Section 11.1, all subtrees are evaluated on training or validation data, and we can analyze the similarity of the values calculated by evaluating the subtrees of the formula trees which are to be compared. We here use validation data for this similarity estimation and the squared differences based approach (using the *sse* function, see Section 11.1 for details).
- Additive structural similarity estimation: Structural components of structure trees are analyzed as described in Section 11.2 using the additive approach; we here weight all possible contributing aspects equally, i.e. the contributions'

weighting factors $c_{1...10}$ are all set to 1.0, only the level difference is weighted stronger with factor 4.0.

- Multiplicative structural similarity estimation: Again, structural components of structure trees are analyzed as described in Section 11.2 using the multiplicative approach; again, we set all weighting factors equally, namely to 0.2, only the level difference is weighted stronger with factor 0.8.

Again we have used the NO_x data set III (as characterized in Section 14.2.2) and the *Thyroid* data set; in both cases we have selected the first 80% of the given data samples as training data and 400 samples as validation data (which are then used in evaluation based similarity estimation). Thus, the four test cases which are summarized in Table 17.25 are formed.

Table 17.25: GP test strategies for similarity estimators comparison.

Test Case	*GP Strategy*	*Data Set*
I	EGP	NO_x
II	EGP	*Thyroid*
III	SGP	NO_x
IV	SGP	*Thyroid*

All test cases were executed three times independently; the maximum tree height was set to 6, the maximum tree size to 50 (for NO_x as well as *Thyroid* tests). The similarity values among individuals were calculated in the context of population diversity estimation analysis executed after every 100th generation in SGP runs and after each 5th generation in extended GP (EGP) runs.

17.5.2 Test Results

What we are most interested in here is not how the GP strategies affect the population diversity during the GP executions, but we rather want to document the relationship between the similarity values calculated using the two approaches chosen. So we have collected the results of all similarity calculations; as this is done for 1,000 models we get 1,000,000 for each similarity function each time the population is analyzed. For each SGP test we therefore eventually get 21 million similarity values for each function (because we also analyze after initializing the population), and for each EGP test we get a comparable amount of similarity values[4].

As we do here not care about differences between EGP and SGP we have collected all similarity calculation results for the NO_x and the *Thyroid* runs separately; the NO_x series are hereafter referred to as series *(n)*, the *Thyroid* runs as *(t)*.

The similarity values calculated for the *(n)* series using evaluation based, additive structural and multiplicative structural comparison are hereafter denoted as $\mathbf{n_e}$, $\mathbf{n_{s1}}$ and $\mathbf{n_{s2}}$, respectively; in analogy to this, the similarity values for the *(t)* series are denoted as $\mathbf{t_e}$, $\mathbf{t_{s1}}$ and $\mathbf{t_{s2}}$, respectively. Please note that for each index i the values $n_e(i)$, $n_{s1}(i)$ and $n_{s2}(i)$ belong to the same pair of models (structure trees) that have been compared; in analogy to this, for each index i also the corresponding comparison results $t_e(i)$, $t_{s1}(i)$ and $t_{s2}(i)$ are associated to the same pair of formulas.

All test runs were executed on Pentium© 4 computers with 3.00 GHz CPU speed and 2 GB RAM.

First, several statistics are calculated for the similarity values collected in $\mathbf{n_e}$, $\mathbf{n_{s1}}$, $\mathbf{n_{s2}}$, $\mathbf{t_e}$, $\mathbf{t_{s1}}$ and $\mathbf{t_{s2}}$; N_n stands for the number of values in $\mathbf{n_e}$, $\mathbf{n_{s1}}$ and $\mathbf{n_{s2}}$, N_t for the number of values in $\mathbf{t_e}$, $\mathbf{t_{s1}}$ and $\mathbf{t_{s2}}$. The results are summarized in Table 17.26; *std* here stands for standard deviation ($std(\mathbf{x}) = \sqrt{\frac{1}{N}\sum_{i\in[1;N]}(x_i - \bar{x})^2}, \bar{x} = \frac{1}{N}\sum_{i\in[1;N]} x, N = |x|$), and *corr* again for the linear correlation (please see for example Section 11.1 for details about this function).

Obviously, the structural similarity values tend to be a lot higher than the evaluation based ones - which is not really surprising as even small changes in the formula's structure can affect its evaluation significantly. The mean squared difference between structural and evaluation based similarity values ranges from ~0.08 to ~0.12; the respective standard deviations of the similarity differences range from 0.15 to ~0.216. The much more informative statistic feature is the linear correlation coefficient: Analyzing NO_x tests we see that the correlation between structural and evaluation based similarities is between ~0.82 (for the additive structural calculation) and ~0.8455 (for multiplicative structural approach); for the *Thyroid* tests, these are not quite as high, namely ~0.76 and ~0.8, respectively.

As we had expected, the correlation between the results calculated using the additive structural model comparison method and the multiplicative one is very high, namely approximately 0.995 for NO_x as well as *Thyroid* tests.

The runtime consumption of the evaluation based similarity estimation method

[4]This number is not constant for EGP due to the fact that the selection pressure reaches its limit not at the same time in each test case execution.

Table 17.26: Comparing similarity estimation results: Basic statistics.

$mean(\mathbf{n_e}) = \frac{1}{N_n}\sum_{i\in[1;N_n]}(n_e(i))$	0.3444
$mean(\mathbf{n_{s1}}) = \frac{1}{N_n}\sum_{i\in[1;N_n]}(n_{s1}(i))$	0.6467
$mean(\mathbf{n_{s2}}) = \frac{1}{N_n}\sum_{i\in[1;N_n]}(n_{s2}(i))$	0.6061
$mean(\mathbf{t_e}) = \frac{1}{N_n}\sum_{i\in[1;N_t]}(t_e(i))$	0.4224
$mean(\mathbf{t_{s1}}) = \frac{1}{N_n}\sum_{i\in[1;N_t]}(t_{s1}(i))$	0.6595
$mean(\mathbf{t_{s2}}) = \frac{1}{N_n}\sum_{i\in[1;N_t]}(t_{s2}(i))$	0.6327
$mse(\mathbf{n_e}, \mathbf{n_{s1}}) = \frac{1}{N_n}\sum_{i\in[1;N_n]}(n_e(i) - n_{s1}(i))^2$	0.1178
$mse(\mathbf{n_e}, \mathbf{n_{s2}}) = \frac{1}{N_n}\sum_{i\in[1;N_n]}(n_e(i) - n_{s2}(i))^2$	0.0910
$mse(\mathbf{n_{s1}}, \mathbf{n_{s2}}) = \frac{1}{N_n}\sum_{i\in[1;N_n]}(n_{s1}(i) - n_{s2}(i))^2$	0.0024
$mse(\mathbf{t_e}, \mathbf{t_{s1}}) = \frac{1}{N_n}\sum_{i\in[1;N_t]}(t_e(i) - t_{s1}(i))^2$	0.1028
$mse(\mathbf{t_e}, \mathbf{t_{s2}}) = \frac{1}{N_n}\sum_{i\in[1;N_n]}(t_e(i) - t_{s2}(i))^2$	0.0839
$mse(\mathbf{t_{s1}}, \mathbf{t_{s2}}) = \frac{1}{N_n}\sum_{i\in[1;N_n]}(t_{s1}(i) - t_{s2}(i))^2$	0.0016
$std(\mathbf{n_e} - \mathbf{n_{s1}})$	0.1625
$std(\mathbf{n_e} - \mathbf{n_{s2}})$	0.1500
$std(\mathbf{n_{s1}} - \mathbf{n_{s2}})$	0.0268
$std(\mathbf{t_e} - \mathbf{t_{s1}})$	0.2159
$std(\mathbf{t_e} - \mathbf{t_{s2}})$	0.1992
$std(\mathbf{t_{s1}} - \mathbf{t_{s2}})$	0.0305
$corr(\mathbf{n_e}, \mathbf{n_{s1}})$	0.8179
$corr(\mathbf{n_e}, \mathbf{n_{s2}})$	0.8455
$corr(\mathbf{n_{s1}}, \mathbf{n_{s2}})$	0.9954
$corr(\mathbf{t_e}, \mathbf{t_{s1}})$	0.7634
$corr(\mathbf{t_e}, \mathbf{t_{s2}})$	0.7998
$corr(\mathbf{t_{s1}}, \mathbf{t_{s2}})$	0.9947
Runtime consumption per generation (evaluation based similarity)	2h08'30"
Runtime consumption per generation (structural similarity, per method)	38'02"

is, of course, a lot higher than the runtime consumption caused by structural population diversity analysis: Although only 400 validation samples are evaluated for evaluation based similarity estimation, structural similarity calculation consumes only approximately a fourth as much runtime.

Even more detailed results discussion becomes possible by partitioning all pairs of corresponding similarity values into five groups with equal range. This means that we collect all structural similarity results in the intervals $[0.0 \ldots 0.2]$, $[0.2 \ldots 0.4]$, $\ldots$, $[0.8 \ldots 1.0]$; of course, we also collect all evaluation based similarity values in the same intervals. Thus, what we get is a number of partitions of data sets which are defined and summarized in Table 17.27.

Table 17.27: Partitions formed for detailed comparison of similarity estimation results.

Partition Index	*Index and Data Set Definitions*
$a0$	$I_{a0} = \{i : (0.0 \leq n_e(i) \leq 0.2)\}$; $\mathbf{n_e^{a0}} = \mathbf{n_e}(I_{a0})$, $\mathbf{n_{s1}^{a0}} = \mathbf{n_{s1}}(I_{a0})$, $\mathbf{n_{s2}^{a0}} = \mathbf{n_{s2}}(I_{a0})$
$a1$	$I_{a1} = \{i : (0.2 < n_e(i) \leq 0.4)\}$; $\mathbf{n_e^{a1}} = \mathbf{n_e}(I_{a1})$, $\mathbf{n_{s1}^{a1}} = \mathbf{n_{s1}}(I_{a1})$, $\mathbf{n_{s2}^{a1}} = \mathbf{n_{s2}}(I_{a1})$
$a2$	$I_{a1} = \{i : (0.4 < n_e(i) \leq 0.6)\}$; $\mathbf{n_e^{a2}} = \mathbf{n_e}(I_{a2})$, $\mathbf{n_{s1}^{a2}} = \mathbf{n_{s1}}(I_{a2})$, $\mathbf{n_{s2}^{a2}} = \mathbf{n_{s2}}(I_{a2})$
$a3$	$I_{a1} = \{i : (0.6 < n_e(i) \leq 0.8)\}$; $\mathbf{n_e^{a3}} = \mathbf{n_e}(I_{a3})$, $\mathbf{n_{s1}^{a3}} = \mathbf{n_{s1}}(I_{a3})$, $\mathbf{n_{s2}^{a3}} = \mathbf{n_{s2}}(I_{a3})$
$a4$	$I_{a1} = \{i : (0.8 < n_e(i) \leq 1.0)\}$; $\mathbf{n_e^{a4}} = \mathbf{n_e}(I_{a4})$, $\mathbf{n_{s1}^{a4}} = \mathbf{n_{s1}}(I_{a4})$, $\mathbf{n_{s2}^{a4}} = \mathbf{n_{s2}}(I_{a4})$
$b0$	$I_{b0} = \{i : (0.0 \leq n_{s1}(i) \leq 0.2)\}$; $\mathbf{n_e^{b0}} = \mathbf{n_e}(I_{b0})$, $\mathbf{n_{s1}^{b0}} = \mathbf{n_{s1}}(I_{b0})$, $\mathbf{n_{s2}^{b0}} = \mathbf{n_{s2}}(I_{b0})$
$b1$	$I_{b1} = \{i : (0.2 < n_{s1}(i) \leq 0.4)\}$; $\mathbf{n_e^{b1}} = \mathbf{n_e}(I_{b1})$, $\mathbf{n_{s1}^{b1}} = \mathbf{n_{s1}}(I_{b1})$, $\mathbf{n_{s2}^{b1}} = \mathbf{n_{s2}}(I_{b1})$
$b2$	$I_{b2} = \{i : (0.4 < n_{s1}(i) \leq 0.6)\}$; $\mathbf{n_e^{b2}} = \mathbf{n_e}(I_{b2})$, $\mathbf{n_{s1}^{b2}} = \mathbf{n_{s1}}(I_{b2})$, $\mathbf{n_{s2}^{b2}} = \mathbf{n_{s2}}(I_{b2})$
$b3$	$I_{b3} = \{i : (0.6 < n_{s1}(i) \leq 0.8)\}$; $\mathbf{n_e^{b3}} = \mathbf{n_e}(I_{b3})$, $\mathbf{n_{s1}^{b3}} = \mathbf{n_{s1}}(I_{b3})$, $\mathbf{n_{s2}^{b3}} = \mathbf{n_{s2}}(I_{b3})$
$b4$	$I_{b4} = \{i : (0.8 < n_{s1}(i) \leq 1.0)\}$; $\mathbf{n_e^{b4}} = \mathbf{n_e}(I_{b4})$, $\mathbf{n_{s1}^{b4}} = \mathbf{n_{s1}}(I_{b4})$, $\mathbf{n_{s2}^{b4}} = \mathbf{n_{s2}}(I_{b4})$
$c0$	$I_{c0} = \{i : (0.0 \leq n_{s2}(i) \leq 0.2)\}$; $\mathbf{n_e^{c0}} = \mathbf{n_e}(I_{c0})$, $\mathbf{n_{s1}^{c0}} = \mathbf{n_{s1}}(I_{c0})$, $\mathbf{n_{s2}^{c0}} = \mathbf{n_{s2}}(I_{c0})$
$c1$	$I_{c1} = \{i : (0.2 < n_{s2}(i) \leq 0.4)\}$; $\mathbf{n_e^{c1}} = \mathbf{n_e}(I_{c1})$, $\mathbf{n_{s1}^{c1}} = \mathbf{n_{s1}}(I_{c1})$, $\mathbf{n_{s2}^{c1}} = \mathbf{n_{s2}}(I_{c1})$
$c2$	$I_{c2} = \{i : (0.4 < n_{s2}(i) \leq 0.6)\}$; $\mathbf{n_e^{c2}} = \mathbf{n_e}(I_{c2})$, $\mathbf{n_{s1}^{c2}} = \mathbf{n_{s1}}(I_{c2})$, $\mathbf{n_{s2}^{c2}} = \mathbf{n_{s2}}(I_{c2})$
$c3$	$I_{c3} = \{i : (0.6 < n_{s2}(i) \leq 0.8)\}$; $\mathbf{n_e^{c3}} = \mathbf{n_e}(I_{c3})$, $\mathbf{n_{s1}^{c3}} = \mathbf{n_{s1}}(I_{c3})$, $\mathbf{n_{s2}^{c3}} = \mathbf{n_{s2}}(I_{c3})$
$c4$	$I_{c4} = \{i : (0.8 < n_{s2}(i) \leq 1.0)\}$; $\mathbf{n_e^{c4}} = \mathbf{n_e}(I_{c4})$, $\mathbf{n_{s1}^{c4}} = \mathbf{n_{s1}}(I_{c4})$, $\mathbf{n_{s2}^{c4}} = \mathbf{n_{s2}}(I_{c4})$
$d0$	$I_{d0} = \{i : (0.0 \leq t_e(i) \leq 0.2)\}$; $\mathbf{t_e^{d0}} = \mathbf{t_e}(I_{d0})$, $\mathbf{t_{s1}^{d0}} = \mathbf{t_{s1}}(I_{d0})$, $\mathbf{t_{s2}^{d0}} = \mathbf{t_{s2}}(I_{d0})$
$d1$	$I_{d1} = \{i : (0.2 < t_e(i) \leq 0.4)\}$; $\mathbf{t_e^{d1}} = \mathbf{t_e}(I_{d1})$, $\mathbf{t_{s1}^{d1}} = \mathbf{t_{s1}}(I_{d1})$, $\mathbf{t_{s2}^{d1}} = \mathbf{t_{s2}}(I_{d1})$
$d2$	$I_{d1} = \{i : (0.4 < t_e(i) \leq 0.6)\}$; $\mathbf{t_e^{d2}} = \mathbf{t_e}(I_{d2})$, $\mathbf{t_{s1}^{d2}} = \mathbf{t_{s1}}(I_{d2})$, $\mathbf{t_{s2}^{d2}} = \mathbf{t_{s2}}(I_{d2})$
$d3$	$I_{d1} = \{i : (0.6 < t_e(i) \leq 0.8)\}$; $\mathbf{t_e^{d3}} = \mathbf{t_e}(I_{d3})$, $\mathbf{t_{s1}^{d3}} = \mathbf{t_{s1}}(I_{d3})$, $\mathbf{t_{s2}^{d3}} = \mathbf{t_{s2}}(I_{d3})$
$d4$	$I_{d1} = \{i : (0.8 < t_e(i) \leq 1.0)\}$; $\mathbf{t_e^{d4}} = \mathbf{t_e}(I_{d4})$, $\mathbf{t_{s1}^{d4}} = \mathbf{t_{s1}}(I_{d4})$, $\mathbf{t_{s2}^{d4}} = \mathbf{t_{s2}}(I_{d4})$
$e0$	$I_{e0} = \{i : (0.0 \leq t_{s1}(i) \leq 0.2)\}$; $\mathbf{t_e^{e0}} = \mathbf{t_e}(I_{e0})$, $\mathbf{t_{s1}^{e0}} = \mathbf{t_{s1}}(I_{e0})$, $\mathbf{t_{s2}^{e0}} = \mathbf{t_{s2}}(I_{e0})$
$e1$	$I_{e1} = \{i : (0.2 < t_{s1}(i) \leq 0.4)\}$; $\mathbf{t_e^{e1}} = \mathbf{t_e}(I_{e1})$, $\mathbf{t_{s1}^{e1}} = \mathbf{t_{s1}}(I_{e1})$, $\mathbf{t_{s2}^{e1}} = \mathbf{t_{s2}}(I_{e1})$
$e2$	$I_{e2} = \{i : (0.4 < t_{s1}(i) \leq 0.6)\}$; $\mathbf{t_e^{e2}} = \mathbf{t_e}(I_{e2})$, $\mathbf{t_{s1}^{e2}} = \mathbf{t_{s1}}(I_{e2})$, $\mathbf{t_{s2}^{e2}} = \mathbf{t_{s2}}(I_{e2})$
$e3$	$I_{e3} = \{i : (0.6 < t_{s1}(i) \leq 0.8)\}$; $\mathbf{t_e^{e3}} = \mathbf{t_e}(I_{e3})$, $\mathbf{t_{s1}^{e3}} = \mathbf{t_{s1}}(I_{e3})$, $\mathbf{t_{s2}^{e3}} = \mathbf{t_{s2}}(I_{e3})$
$e4$	$I_{e4} = \{i : (0.8 < t_{s1}(i) \leq 1.0)\}$; $\mathbf{t_e^{e4}} = \mathbf{t_e}(I_{e4})$, $\mathbf{t_{s1}^{e4}} = \mathbf{t_{s1}}(I_{e4})$, $\mathbf{t_{s2}^{e4}} = \mathbf{t_{s2}}(I_{e4})$
$f0$	$I_{f0} = \{i : (0.0 \leq t_{s2}(i) \leq 0.2)\}$; $\mathbf{t_e^{f0}} = \mathbf{t_e}(I_{f0})$, $\mathbf{t_{s1}^{f0}} = \mathbf{t_{s1}}(I_{f0})$, $\mathbf{t_{s2}^{f0}} = \mathbf{t_{s2}}(I_{f0})$
$f1$	$I_{f1} = \{i : (0.2 < t_{s2}(i) \leq 0.4)\}$; $\mathbf{t_e^{f1}} = \mathbf{t_e}(I_{f1})$, $\mathbf{t_{s1}^{f1}} = \mathbf{t_{s1}}(I_{f1})$, $\mathbf{t_{s2}^{f1}} = \mathbf{t_{s2}}(I_{f1})$
$f2$	$I_{f2} = \{i : (0.4 < t_{s2}(i) \leq 0.6)\}$; $\mathbf{t_e^{f2}} = \mathbf{t_e}(I_{f2})$, $\mathbf{t_{s1}^{f2}} = \mathbf{t_{s1}}(I_{f2})$, $\mathbf{t_{s2}^{f2}} = \mathbf{t_{s2}}(I_{f2})$
$f3$	$I_{f3} = \{i : (0.6 < t_{s2}(i) \leq 0.8)\}$; $\mathbf{t_e^{f3}} = \mathbf{t_e}(I_{f3})$, $\mathbf{t_{s1}^{f3}} = \mathbf{t_{s1}}(I_{f3})$, $\mathbf{t_{s2}^{f3}} = \mathbf{t_{s2}}(I_{f3})$
$f4$	$I_{f4} = \{i : (0.8 < t_{s2}(i) \leq 1.0)\}$; $\mathbf{t_e^{f4}} = \mathbf{t_e}(I_{f4})$, $\mathbf{t_{s1}^{f4}} = \mathbf{t_{s1}}(I_{f4})$, $\mathbf{t_{s2}^{f4}} = \mathbf{t_{s2}}(I_{f4})$

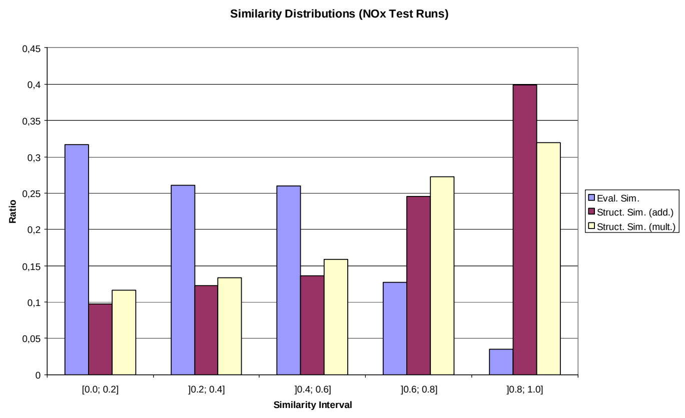
Now we can analyze these partitions separately: For each partition we have calculated the linear correlation between evaluation based, additive structural and multiplicative structural similarities as well as the mean squared difference between these respective values; Table 17.28 summarizes these partition-wise statistics. Additionally, the frequency of each partition is also given: The frequency of a partition is hereby given by the number of pairs of values included divided by the number of all pairs of values available: $frequ(I_{ki}) = \frac{|I_{ki}|}{\sum_{j\in[0;4]} I_{kj}}$ for $k \in \{a, b, c, d, e, f\}$ and $i \in [0; 4]$.

Figure 17.23 shows the distributions of structural and evaluation based similarity estimation for the NO_x and *Thyroid* tests separately. As we see in both charts the structural similarity values are significantly higher than the evaluation based ones.

Table 17.28: Comparing similarity estimation results: Detailed partition-wise statistics.

$freq(I_{a0}) = 0.3172$	$corr(\mathbf{n_{e}^{a0}}, \mathbf{n_{s1}^{a0}}) = 0.6294$ $mse(\mathbf{n_{e}^{a0}}, \mathbf{n_{s1}^{a0}}) = 0.1061$	$corr(\mathbf{n_{e}^{a0}}, \mathbf{n_{s2}^{a0}}) = 0.6772$ $mse(\mathbf{n_{e}^{a0}}, \mathbf{n_{s2}^{a0}}) = 0.0751$
$freq(I_{a1}) = 0.2609$	$corr(\mathbf{n_{e}^{a1}}, \mathbf{n_{s1}^{a1}}) = 0.8407$ $mse(\mathbf{n_{e}^{a1}}, \mathbf{n_{s1}^{a1}}) = 0.1083$	$corr(\mathbf{n_{e}^{a1}}, \mathbf{n_{s2}^{a1}}) = 0.8574$ $mse(\mathbf{n_{e}^{a1}}, \mathbf{n_{s2}^{a1}}) = 0.0818$
$freq(I_{a2}) = 0.2595$	$corr(\mathbf{n_{e}^{a2}}, \mathbf{n_{s1}^{a2}}) = 0.7886$ $mse(\mathbf{n_{e}^{a2}}, \mathbf{n_{s1}^{a2}}) = 0.1364$	$corr(\mathbf{n_{e}^{a2}}, \mathbf{n_{s2}^{a2}}) = 0.8047$ $mse(\mathbf{n_{e}^{a2}}, \mathbf{n_{s2}^{a2}}) = 0.1106$
$freq(I_{a3}) = 0.1272$	$corr(\mathbf{n_{e}^{a3}}, \mathbf{n_{s1}^{a3}}) = 0.6963$ $mse(\mathbf{n_{e}^{a3}}, \mathbf{n_{s1}^{a3}}) = 0.1279$	$corr(\mathbf{n_{e}^{a3}}, \mathbf{n_{s2}^{a3}}) = 0.7376$ $mse(\mathbf{n_{e}^{a3}}, \mathbf{n_{s2}^{a3}}) = 0.1077$
$freq(I_{a4}) = 0.0352$	$corr(\mathbf{n_{e}^{a4}}, \mathbf{n_{s1}^{a4}}) = 0.7174$ $mse(\mathbf{n_{e}^{a4}}, \mathbf{n_{s1}^{a4}}) = 0.1184$	$corr(\mathbf{n_{e}^{a4}}, \mathbf{n_{s2}^{a4}}) = 0.7559$ $mse(\mathbf{n_{e}^{a4}}, \mathbf{n_{s2}^{a4}}) = 0.0983$
$freq(I_{b0}) = 0.0974$	$corr(\mathbf{n_{s1}^{b0}}, \mathbf{n_{e}^{b0}}) = 0.3815$ $mse(\mathbf{n_{s1}^{b0}}, \mathbf{n_{e}^{b0}}) = 0.1407$	$corr(\mathbf{n_{s1}^{b0}}, \mathbf{n_{s2}^{b0}}) = 0.9890$ $mse(\mathbf{n_{s1}^{b0}}, \mathbf{n_{s2}^{b0}}) = 0.0057$
$freq(I_{b1}) = 0.1222$	$corr(\mathbf{n_{s1}^{b1}}, \mathbf{n_{e}^{b1}}) = 0.6744$ $mse(\mathbf{n_{s1}^{b1}}, \mathbf{n_{e}^{b1}}) = 0.0884$	$corr(\mathbf{n_{s1}^{b1}}, \mathbf{n_{s2}^{b1}}) = 0.9931$ $mse(\mathbf{n_{s1}^{b1}}, \mathbf{n_{s2}^{b0}}) = 0.0028$
$freq(I_{b2}) = 0.1363$	$corr(\mathbf{n_{s1}^{b2}}, \mathbf{n_{e}^{b2}}) = 0.7591$ $mse(\mathbf{n_{s1}^{b2}}, \mathbf{n_{e}^{b2}}) = 0.0985$	$corr(\mathbf{n_{s1}^{b2}}, \mathbf{n_{s2}^{b2}}) = 0.9962$ $mse(\mathbf{n_{s1}^{b2}}, \mathbf{n_{s2}^{b0}}) = 0.0026$
$freq(I_{b3}) = 0.2451$	$corr(\mathbf{n_{s1}^{b3}}, \mathbf{n_{e}^{b3}}) = 0.8350$ $mse(\mathbf{n_{s1}^{b3}}, \mathbf{n_{e}^{b3}}) = 0.1080$	$corr(\mathbf{n_{s1}^{b3}}, \mathbf{n_{s2}^{b3}}) = 0.9963$ $mse(\mathbf{n_{s1}^{b3}}, \mathbf{n_{s2}^{b0}}) = 0.0024$
$freq(I_{b4}) = 0.3990$	$corr(\mathbf{n_{s1}^{b4}}, \mathbf{n_{e}^{b4}}) = 0.7677$ $mse(\mathbf{n_{s1}^{b4}}, \mathbf{n_{e}^{b4}}) = 0.1337$	$corr(\mathbf{n_{s1}^{b4}}, \mathbf{n_{s2}^{b4}}) = 0.9975$ $mse(\mathbf{n_{s1}^{b4}}, \mathbf{n_{s2}^{b0}}) = 0.0013$
$freq(I_{c0}) = 0.1160$	$corr(\mathbf{n_{s2}^{c0}}, \mathbf{n_{e}^{c0}}) = 0.4119$ $mse(\mathbf{n_{s2}^{c0}}, \mathbf{n_{e}^{c0}}) = 0.0997$	$corr(\mathbf{n_{s2}^{c0}}, \mathbf{n_{s1}^{c0}}) = 0.9888$ $mse(\mathbf{n_{s2}^{c0}}, \mathbf{n_{s1}^{c0}}) = 0.0059$
$freq(I_{c1}) = 0.1335$	$corr(\mathbf{n_{s2}^{c1}}, \mathbf{n_{e}^{c1}}) = 0.7667$ $mse(\mathbf{n_{s2}^{c1}}, \mathbf{n_{e}^{c1}}) = 0.0580$	$corr(\mathbf{n_{s2}^{c1}}, \mathbf{n_{s1}^{c1}}) = 0.9961$ $mse(\mathbf{n_{s2}^{c1}}, \mathbf{n_{s1}^{c0}}) = 0.0023$
$freq(I_{c2}) = 0.1584$	$corr(\mathbf{n_{s2}^{c2}}, \mathbf{n_{e}^{c2}}) = 0.8229$ $mse(\mathbf{n_{s2}^{c2}}, \mathbf{n_{e}^{c2}}) = 0.0730$	$corr(\mathbf{n_{s2}^{c2}}, \mathbf{n_{s1}^{c2}}) = 0.9963$ $mse(\mathbf{n_{s2}^{c2}}, \mathbf{n_{s1}^{c0}}) = 0.0027$
$freq(I_{c3}) = 0.2728$	$corr(\mathbf{n_{s2}^{c3}}, \mathbf{n_{e}^{c3}}) = 0.8764$ $mse(\mathbf{n_{s2}^{c3}}, \mathbf{n_{e}^{c3}}) = 0.0794$	$corr(\mathbf{n_{s2}^{c3}}, \mathbf{n_{s1}^{c3}}) = 0.9967$ $mse(\mathbf{n_{s2}^{c3}}, \mathbf{n_{s1}^{c0}}) = 0.0021$
$freq(I_{c4}) = 0.3193$	$corr(\mathbf{n_{s2}^{c4}}, \mathbf{n_{e}^{c4}}) = 0.7528$ $mse(\mathbf{n_{s2}^{c4}}, \mathbf{n_{e}^{c4}}) = 0.1205$	$corr(\mathbf{n_{s2}^{c4}}, \mathbf{n_{s1}^{c4}}) = 0.9969$ $mse(\mathbf{n_{s2}^{c4}}, \mathbf{n_{s1}^{c0}}) = 0.0011$
$freq(I_{d0}) = 0.3241$	$corr(\mathbf{t_{e}^{d0}}, \mathbf{t_{s1}^{d0}}) = 0.4233$ $mse(\mathbf{t_{e}^{d0}}, \mathbf{t_{s1}^{d0}}) = 0.1964$	$corr(\mathbf{t_{e}^{d0}}, \mathbf{t_{s2}^{d0}}) = 0.4777$ $mse(\mathbf{t_{e}^{d0}}, \mathbf{t_{s2}^{d0}}) = 0.1572$
$freq(I_{d1}) = 0.1239$	$corr(\mathbf{t_{e}^{d1}}, \mathbf{t_{s1}^{d1}}) = 0.8323$ $mse(\mathbf{t_{e}^{d1}}, \mathbf{t_{s1}^{d1}}) = 0.0336$	$corr(\mathbf{t_{e}^{d1}}, \mathbf{t_{s2}^{d1}}) = 0.8409$ $mse(\mathbf{t_{e}^{d1}}, \mathbf{t_{s2}^{d1}}) = 0.0295$
$freq(I_{d2}) = 0.2216$	$corr(\mathbf{t_{e}^{d2}}, \mathbf{t_{s1}^{d2}}) = 0.8455$ $mse(\mathbf{t_{e}^{d2}}, \mathbf{t_{s1}^{d2}}) = 0.0703$	$corr(\mathbf{t_{e}^{d2}}, \mathbf{t_{s2}^{d2}}) = 0.8606$ $mse(\mathbf{t_{e}^{d2}}, \mathbf{t_{s2}^{d2}}) = 0.0587$
$freq(I_{d3}) = 0.1919$	$corr(\mathbf{t_{e}^{d3}}, \mathbf{t_{s1}^{d3}}) = 0.8471$ $mse(\mathbf{t_{e}^{d3}}, \mathbf{t_{s1}^{d3}}) = 0.0688$	$corr(\mathbf{t_{e}^{d3}}, \mathbf{t_{s2}^{d3}}) = 0.8607$ $mse(\mathbf{t_{e}^{d3}}, \mathbf{t_{s2}^{d3}}) = 0.0566$
$freq(I_{d4}) = 0.1385$	$corr(\mathbf{t_{e}^{d4}}, \mathbf{t_{s1}^{d4}}) = 0.7956$ $mse(\mathbf{t_{e}^{d4}}, \mathbf{t_{s1}^{d4}}) = 0.0433$	$corr(\mathbf{t_{e}^{d4}}, \mathbf{t_{s2}^{d4}}) = 0.8109$ $mse(\mathbf{t_{e}^{d4}}, \mathbf{t_{s2}^{d4}}) = 0.0375$
$freq(I_{e0}) = 0.1079$	$corr(\mathbf{t_{s1}^{e0}}, \mathbf{t_{e}^{e0}}) = 0.2693$ $mse(\mathbf{t_{s1}^{e0}}, \mathbf{t_{e}^{e0}}) = 0.1435$	$corr(\mathbf{t_{s1}^{e0}}, \mathbf{t_{s2}^{e0}}) = 0.9853$ $mse(\mathbf{t_{s1}^{e0}}, \mathbf{t_{s2}^{e0}}) = 0.0077$
$freq(I_{e1}) = 0.1043$	$corr(\mathbf{t_{s1}^{e1}}, \mathbf{t_{e}^{e1}}) = 0.3053$ $mse(\mathbf{t_{s1}^{e1}}, \mathbf{t_{e}^{e1}}) = 0.2216$	$corr(\mathbf{t_{s1}^{e1}}, \mathbf{t_{s2}^{e1}}) = 0.9854$ $mse(\mathbf{t_{s1}^{e1}}, \mathbf{t_{s2}^{e0}}) = 0.0024$
$freq(I_{e2}) = 0.1193$	$corr(\mathbf{t_{s1}^{e2}}, \mathbf{t_{e}^{e2}}) = 0.4652$ $mse(\mathbf{t_{s1}^{e2}}, \mathbf{t_{e}^{e2}}) = 0.2120$	$corr(\mathbf{t_{s1}^{e2}}, \mathbf{t_{s2}^{e2}}) = 0.9954$ $mse(\mathbf{t_{s1}^{e2}}, \mathbf{t_{s2}^{e0}}) = 0.0015$
$freq(I_{e3}) = 0.2412$	$corr(\mathbf{t_{s1}^{e3}}, \mathbf{t_{e}^{e3}}) = 0.8559$ $mse(\mathbf{t_{s1}^{e3}}, \mathbf{t_{e}^{e3}}) = 0.0479$	$corr(\mathbf{t_{s1}^{e3}}, \mathbf{t_{s2}^{e3}}) = 0.9986$ $mse(\mathbf{t_{s1}^{e3}}, \mathbf{t_{s2}^{e0}}) = 0.0006$
$freq(I_{e4}) = 0.4274$	$corr(\mathbf{t_{s1}^{e4}}, \mathbf{t_{e}^{e4}}) = 0.8376$ $mse(\mathbf{t_{s1}^{e4}}, \mathbf{t_{e}^{e4}}) = 0.0641$	$corr(\mathbf{t_{s1}^{e4}}, \mathbf{t_{s2}^{e4}}) = 0.9985$ $mse(\mathbf{t_{s1}^{e4}}, \mathbf{t_{s2}^{e0}}) = 0.0006$
$freq(I_{e0}) = 0.1305$	$corr(\mathbf{t_{s2}^{e0}}, \mathbf{t_{e}^{e0}}) = 0.3430$ $mse(\mathbf{t_{s2}^{e0}}, \mathbf{t_{e}^{e0}}) = 0.0837$	$corr(\mathbf{t_{s2}^{e0}}, \mathbf{t_{s1}^{e0}}) = 0.9860$ $mse(\mathbf{t_{s2}^{e0}}, \mathbf{t_{s1}^{e0}}) = 0.0069$
$freq(I_{e1}) = 0.0978$	$corr(\mathbf{t_{s2}^{e1}}, \mathbf{t_{e}^{e1}}) = 0.3380$ $mse(\mathbf{t_{s2}^{e1}}, \mathbf{t_{e}^{e1}}) = 0.2230$	$corr(\mathbf{t_{s2}^{e1}}, \mathbf{t_{s1}^{e1}}) = 0.9964$ $mse(\mathbf{t_{s2}^{e1}}, \mathbf{t_{s1}^{e0}}) = 0.0021$
$freq(I_{e2}) = 0.1456$	$corr(\mathbf{t_{s2}^{e2}}, \mathbf{t_{e}^{e2}}) = 0.6722$ $mse(\mathbf{t_{s2}^{e2}}, \mathbf{t_{e}^{e2}}) = 0.1307$	$corr(\mathbf{t_{s2}^{e2}}, \mathbf{t_{s1}^{e2}}) = 0.9960$ $mse(\mathbf{t_{s2}^{e2}}, \mathbf{t_{s1}^{e0}}) = 0.0012$
$freq(I_{e3}) = 0.2435$	$corr(\mathbf{t_{s2}^{e3}}, \mathbf{t_{e}^{e3}}) = 0.8513$ $mse(\mathbf{t_{s2}^{e3}}, \mathbf{t_{e}^{e3}}) = 0.0505$	$corr(\mathbf{t_{s2}^{e3}}, \mathbf{t_{s1}^{e3}}) = 0.9986$ $mse(\mathbf{t_{s2}^{e3}}, \mathbf{t_{s1}^{e0}}) = 0.0007$
$freq(I_{e4}) = 0.3826$	$corr(\mathbf{t_{s2}^{e4}}, \mathbf{t_{e}^{e4}}) = 0.8501$ $mse(\mathbf{t_{s2}^{e4}}, \mathbf{t_{e}^{e4}}) = 0.0518$	$corr(\mathbf{t_{s2}^{e4}}, \mathbf{t_{s1}^{e4}}) = 0.9985$ $mse(\mathbf{t_{s2}^{e4}}, \mathbf{t_{s1}^{e0}}) = 0.0005$

Regarding results correlations, the figures documented in Table 17.28 can be summarized in the following way: The correlations between structural and evalua-



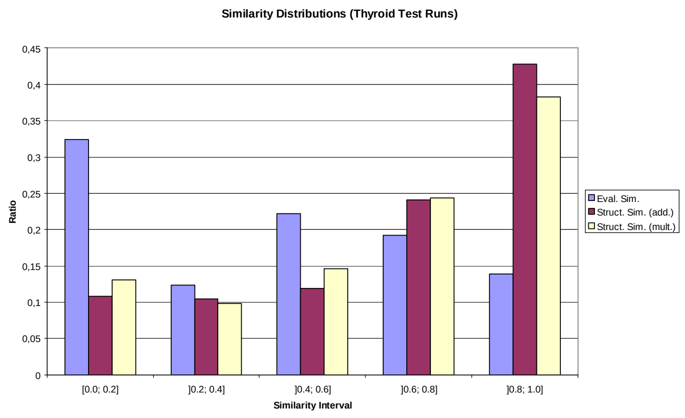


Figure 17.23: Distribution of similarity values calculated using structural and evaluation based similarity functions.

tion based similarity is approximately in the range between 0.3 and 0.85. Especially low correlation coefficients are calculated for the comparison of structural and evaluation based similarities, especially when the structural similarity is considered very low (<0.4). This impression becomes even more clear when we analyze Figures 17.24 and 17.25 which give the partition wise correlations of similarity values. In each of the 6 series given in these Figures we show the correlations of similarity values calculated by each possible pair of methods; in each case those partitions of value pairs are selected that correspond to the values calculated by the first method mentioned in the respective label. So, for example, in the first series we see the partition-wise correlations of similarity values calculated by the evaluation based and the additive structural method; the values are classified in partitions with respect to the evaluation specific similarities.
The Figures 17.24 and 17.25 show clearly that the structural similarity estimation methods calculate very similar values (with high correlations for trees that are very different as well as for those which are considered rather similar). Furthermore, the

correlation of structural and evaluation based similarity values is rather low in the case of low structural similarities (<0.4).

Finally, for graphically illustrating the direct comparison of similarity values calculated by the three estimation methods chosen we have randomly chosen 100,000 structure tree comparison cases both from the NO_x and the *Thyroid* tests. The respectively correspondent similarity values are drawn against each other in the Figures 17.26 – 17.28. On the one hand there is no high correlation which can be seen when comparing structural and evaluation based similarity values, but on the other hand the high correlation between the similarities calculated by the structural similarity estimation methods becomes obvious.

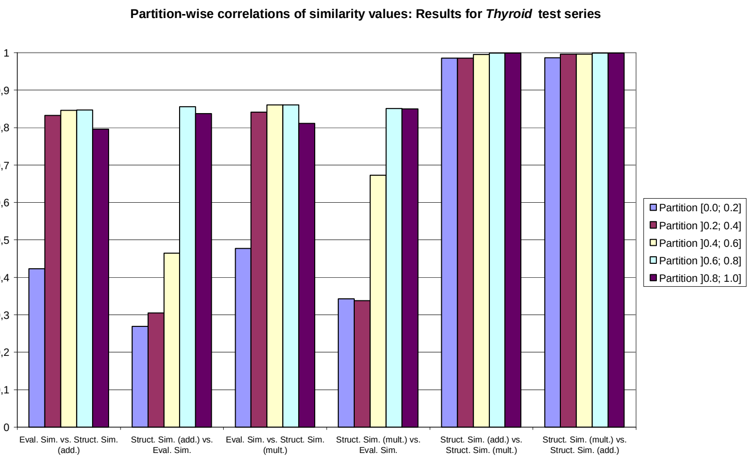


Figure 17.24: Partition-wise correlations of similarity values for NO_x test series.

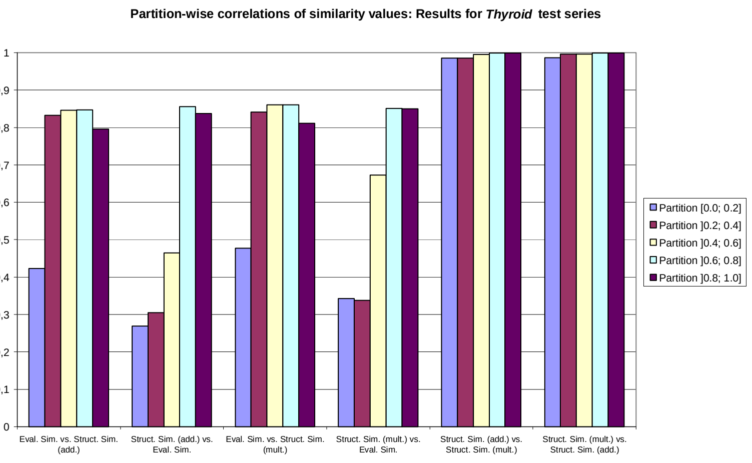
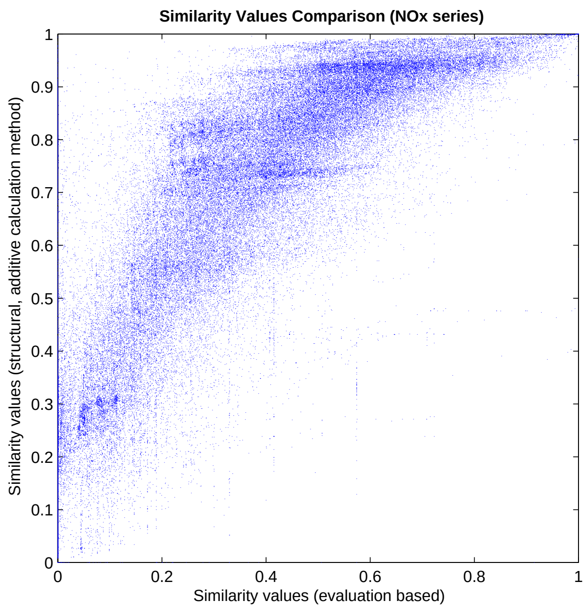
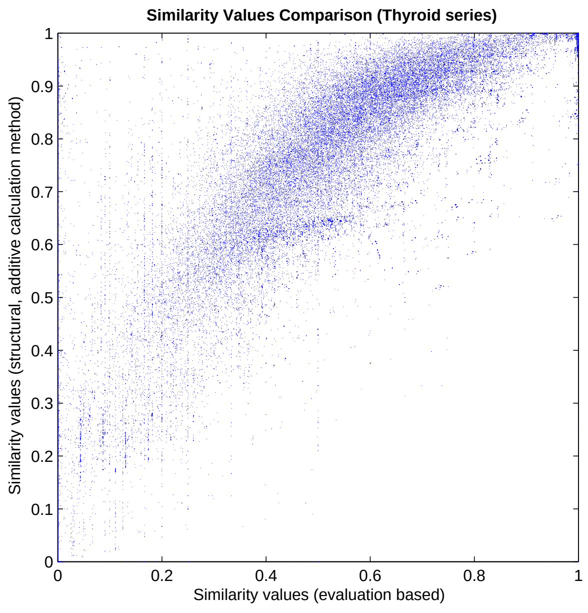


Figure 17.25: Partition-wise correlations of similarity values for *Thyroid* test series.

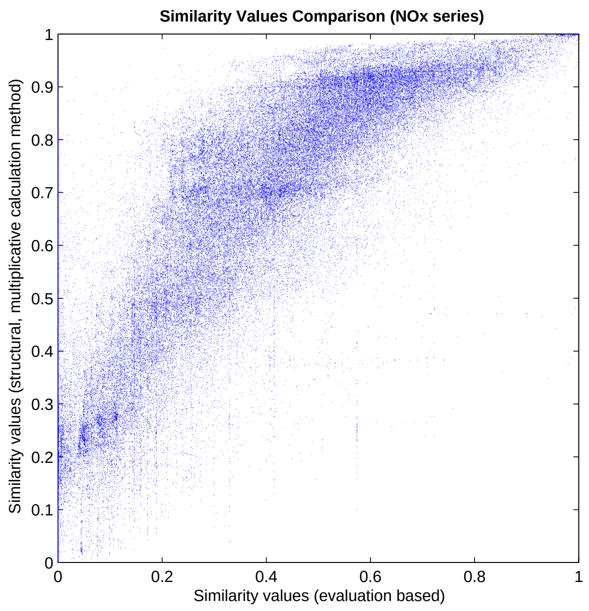


(a) NO_x test series

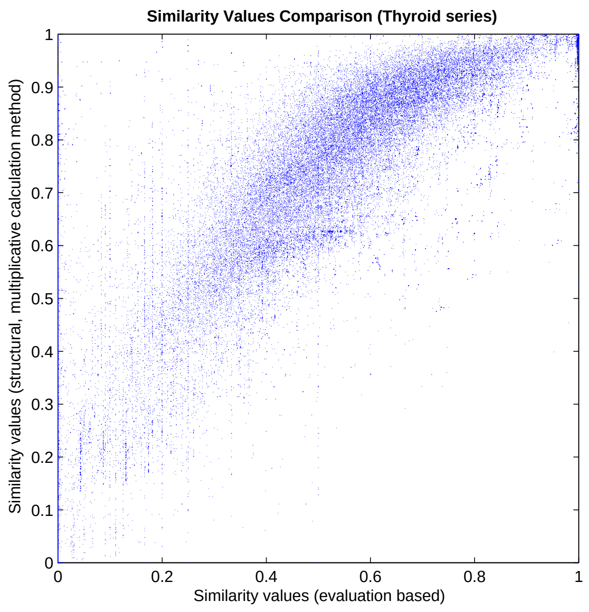


(b) *Thyroid* test series

Figure 17.26: Similarity values comparison: Evaluation based vs. structural (additive calculation).

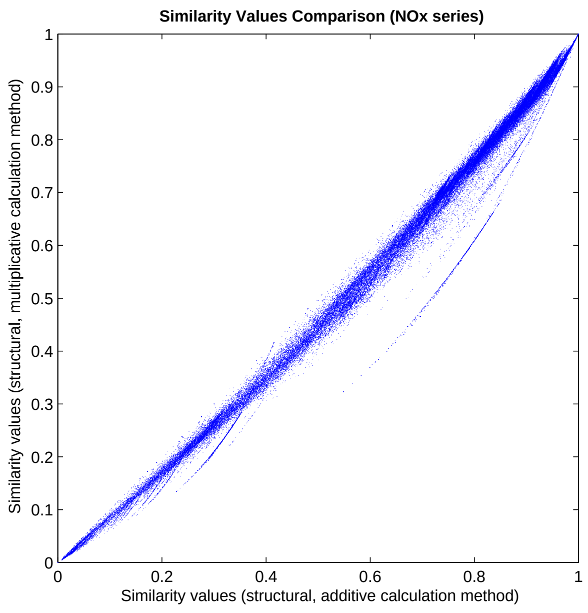


(a) NO_x test series

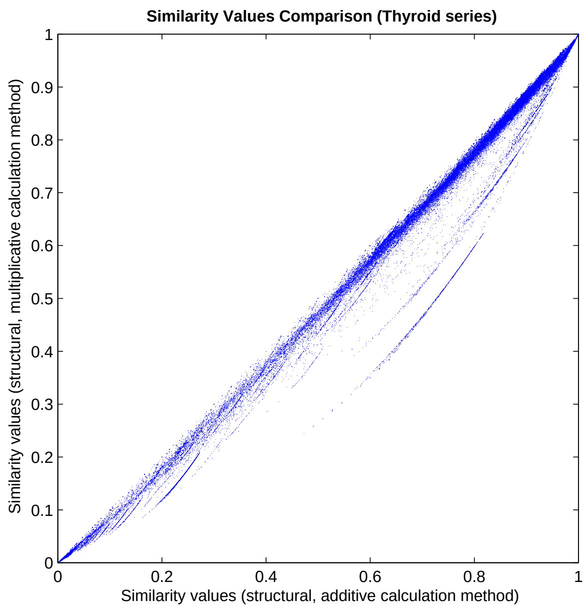


(b) *Thyroid* test series

Figure 17.27: Similarity values comparison: Evaluation based vs. structural (multiplicative calculation).



(a) *NO_x test series*



(b) *Thyroid* test series

Figure 17.28: Similarity values comparison: Structural (additive calculation) vs. structural (multiplicative calculation).

17.5.3 Conclusion

In this section we have summarized a series of GP test runs incorporating evaluation based as well as structural similarity estimation functions for measuring the genetic diversity in GP populations.

In general, evaluation based similarity calculation consumes a lot more runtime than structural comparison, and on average it also tends to produce lower similarity values. The results show that in most cases there is a linear correlation of approximately $0.4 - 0.9$ for the results returned by the evaluation based and structural methods; not very surprisingly, this correlation is positive, but not very high. Especially in some cases showing very low structural similarity there can be significantly different results when using the evaluation based similarity methods.

Furthermore, we have also compared additive and multiplicative structural similarity estimation. These two variants tend to produce rather similar results with high correlations for pairs of structure trees with low as well as rather high similarities; the results retrieved by the multiplicative structural method show a higher correlation with those calculated using the evaluation based similarity function.

17.6 Code Bloat, Pruning, and Population Diversity

17.6.1 Introduction

In Chapter 3.6 we have described one of the major problems of genetic programming, namely permanent code growth, often also referred to as bloat; evolution is also seen as "survival of the fattest", and, as Langdon and Poli expressed it, fitness based selection leads to the fact that "fitness causes bloat" [LP97]. Several approaches for combating this unwanted unlimited growth of chromosome size, some of them being

- limiting the size and / or the height of the program trees,
- pruning programs, and
- punishing complex programs by decreasing their quality depending on their respective tree representations' size and / or height.

Of course, there is no optimal strategy for fixing formula size parameters, population size or pruning strategies a priori (see also remarks in Chapter 3). Still, some code prevention strategies are surely more recommendable than others; we here report on an exemplary test series for characterizing some of the possible approaches.

In all other test series executed and reported on in other sections in this thesis we have used fixed complexity limits (limiting size and height of program trees); we shall here report on our tests regarding code growth in GP based structure identification applying the pruning strategies presented in Section 10.2 as well as structure tree size dependent fitness manipulation and fixed size limits (partially with additional pruning). All these approaches have been tested using standard GP as well as extended GP including gender specific selection and offspring selection. As an example, we have tested these GP variants on the NO_x data set II presented and described in Section 14.2; population diversity, formula complexity parameters as well as additional pruning effort (only in case of applying pruning, of course) have been monitored and shall be reported on here.

We have again used 50% of the given data for training models (namely samples 10,000 – 28,000), and 10% as validation data (samples 28,001 – 32,000 used by pruning strategies) and ~7.5% as test data (samples 32,001 – 35,000). As we are also aware of the problem of overfitting, we have systematically collected each GP run's best models with respect to best fit on training as well as on validation data (using the *mse* function for estimating the formulas' qualities); the algorithm is designed to optimize formulas with respect to training data, validation data are only used for pruning strategies (if used at all). At the end of each test run, the models with best fit on training as well as on validation data are analyzed, and in order to fight overfitting we select the best model on validation data as the result returned by the algorithm. Test data, which are not available to the algorithm, are used for demonstrating that this strategy is a reasonable one: Analyzing the evaluation of the best models on test data we see that those that are best on validation data perform better on test data than those that were optimally fit to training data.

During the GP process, the standard mean squared error function (with early abortion as described in 8.1.3) was used; the time series specific fitness function considering plain values as well as differential and integral values was used for selecting those models that perform best on training and validation data. All three components (i.e., plain values, differentials and integral values) have been weighted using equal weighting factors. When comparing the quality of the results documented in the following sections we again state the fitness values calculated using the mean squared errors function; the maximum punishment factor was set to 10.0.

17.6.2 Test Strategies

In detail, the following test strategies have been applied: On the one hand the parameters for standard and extended GP are summarized in Table 17.29, the code growth prevention parameters are summarized in Table 17.30. In all tests the initial population was created using a size limit of 50 nodes and a maximum height of 6 levels for each structure tree.

Table 17.29: GP parameters used for code growth and bloat prevention tests.

Variant	*Parameters*
1 *(Standard GP, SGP)*	Population size: 1000 2000 generations Single point crossover; structural and parametric node mutation Parents selection: Tournament selection ($k = 3$)
2 *(Extended GP, EGP)*	Population size: 1000 Single point crossover; structural and parametric node mutation Parents selection: Gender specific selection (random & proportional) Strict offspring selection; maximum selection pressure: 100

In the following table and in the explanations given afterwards, md is the maximum deterioration limit and mc the maximum coefficient of deterioration and structure complexity reduction as described in Section 10.2. For ES-based pruning, mr denotes the maximum number of rounds, and mur the maximum number of unsuccessful rounds.

In those tests including increased pruning (as applied in test series (h) and (i)) the initial pruning ratio is set to 0.3, i.e. in the beginning 30% of the population are pruned. Then, during the process execution, the pruning rate steadily increases and finally reaches 0.8; in standard GP runs, the rate is increased linearly, in extended GP including offspring selection we compute the actual pruning ratio in relation to the actual selection pressure (so that in the end, when the selection pressure has reached its maximum value, the pruning rate has also reached its maximum, namely 0.8). Furthermore, fs stands for the formula's size (i.e., the number of nodes in the corresponding structure tree), and pf is the fitness punishment factor: If structure complexity based punishment is applied, then the fitness f of a model is modified as $f' = f * (1 + pf)$ (if $pf > 0$).

Table 17.30: Summary of the code growth prevention strategies applied in these test series.

Variant	*Characteristics*
a	No code growth prevention strategy
b	20% systematic pruning: $md = 0$, $mc = 1$
c	20% ES-based pruning: $md = 0$, $mc = 1$, $\lambda = 5$, $mr = 5$, $mur = 1$
d	50% ES-based pruning: $md = 0.5$, $mc = 1$, $\lambda = 10$, $mr = 10$, $mur = 2$
e	100% ES-based pruning: $md = 2$, $mc = 1.5$, $\lambda = 20$, $mr = 10$, $mur = 2$
f	Increasing ES-based pruning: $md = 1$, $mc = 1.5$, $\lambda = 10$, $mr = 10$, $mur = 2$
g	Quality punishment: $pf = (fs - 50)/50$
h	Fixed limits: Maximum tree height 6, maximum tree size 50
i	Fixed limits: Maximum tree height 6, maximum tree size 50 combined with occasional ES-based pruning standard GP: every 5^{th}, extended GP: every 2^{nd} generation $md = 1$, $mc = 1$, $\lambda = 10$, $mr = 5$, $mur = 2$

Please note that in strategies (b) and (c) pruning is done after each generation step, whereas in (d) – (g) it is done after each creation of a new model by crossover and / or mutation. In standard GP this does not make any difference, but when using offspring selection the decision whether to prune after each creation or after each generation has major effects on the algorithmic process.

The mean squared errors function (with early stopping, see Section 7.5.5) was used here since we mainly concentrate on pruning and population dynamics relevant aspects. Furthermore, all variables (including the target variable) were linearly scaled to the interval [-100; +100].

17.6.3 Test Results

Once again, all test strategies have been executed 5 times independently; formula complexity has been monitored (and protocolled after each generation step) as well as structural population diversity which was protocolled after every 10^{th} generation: The multiplicative similarity approach (as defined in Equations 11.21 – 11.23) has again been chosen, all coefficients $c_1 \dots c_{10}$ were set to 0.2, only the coefficient c_1 weighting the level similarity contribution s_1 was set to 0.8. The similarity of models was calculated symmetrically (as described in Equation 12.48).

17.6.3.1 No Formula Size Limitation

Exactly as we had expected, extreme code growth also occurs in GP-based structure identification; Figure 17.29 illustrates the progress of formula complexity in terms of formula size in exemplary test runs of series 1a and 2a: The average formula size is given as well as minimum and maximum values and the progress of the best individual's size.
As we see here, formulas tend to grow very big rather quickly; when using offspring selection, this effect is even a bit more obvious: On average, in standard GP the formula size has reached 212.84 after 30 iterations, when using OS the average formula size was even higher after 30 generations (namely 276.35).

17.6.3.2 Light Pruning

The results of test series (b) and (c) can be summarized in the following way: Without any further mechanisms that limit the structural complexity of formula trees, light pruning as described in strategies (b) and (c) is not an appropriate way to prevent GP from growing enormous formula structures. After 100 generations, the average formula size in standard GP has grown to 471.34 in test series (1b) and 333.65 in test runs of series (1c) (average standard deviation: 204.29 and 238.27, respectively); in extended GP the average formula size at generation 30 on average reached 293.26 and 276.12 in test runs (2b) and (2c), the respective standard

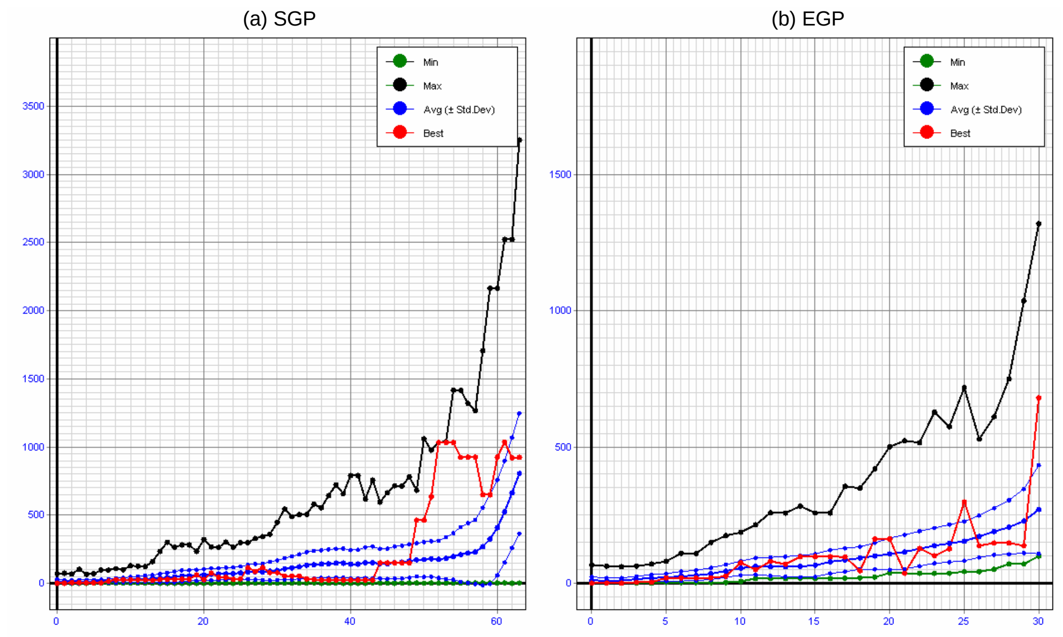


Figure 17.29: Code growth in GP without applying size limits or complexity punishment strategies (left: standard GP, right: extended GP).

deviations being 157.23 and 124.80.

Systematically analyzing the results of the pruning phases performed in test runs (b) and (c) we can compare the performances of ES-based and systematic pruning. For this purpose we have collected the pruning performance statistics for the tests (b) and (c) and summarize them in Table 17.31:

Table 17.31: Performance of systematic and ES-based pruning strategies.

Parameter	*Systematic pruning*	*ES-based pruning*
Solutions evaluated for pruning one solution	161.02	54.56
Runtime consumed (per iteration)	31.27 sec	12.23 sec
Average coefficient of deterioration and reduction of structural complexity	0.2495	0.4053

Obviously, both pruning methods performed approximately equally well and were able to reduce the complexity of the formulas that were supposed to be pruned. Additionally, we also see that especially for bigger model structures the runtime consumption is a lot higher when using systematic pruning; in the course of a GP process it is not considered necessary or even beneficial to reduce models as much as possible, therefore we shall in the following test runs concentrate on ES-based pruning phases. Thus, we suggest using systematic pruning as a preparation step for results analysis, but not during the execution of GP based training processes.

17.6.3.3 Medium Pruning

Medium pruning, as applied in test series (d), is in fact able to reduce the size of the formulas stored in the GP populations significantly.

Table 17.32: Formula size progress in test series (d).

Test series	*Iteration*	*Formula size*	
		avg	*std*
(1d)	20	21.83	32.12
	50	74.24	111.84
	100	123.67	144.78
	500	167.51	156.89
	2000	168.23	147.56
(2d)	10	10.77	13.27
	20	90.43	52.79
	50	228.02	112.51
	End of run	283.98	172.33

Table 17.33: Quality of results returned in test series (d).

Test series	*Evaluation data*	*Best model selection basis*			
		Training data		*Validation data*	
		avg	*std*	*avg*	*std*
(1d)	Training data	1,178.13	205.20	8,231.38	1,041.87
	Validation data	17,962.78	762.97	15,850.49	1,309.10
	Test data	7,162.48	690.10	5,996.27	927.09
(2d)	Training data	1,823.43	823.56	6,005.74	729.47
	Validation data	14,590.83	1,476.25	10,506.30	981.35
	Test data	6,341.28	770.42	4,439.27	918.72

The best results obtained in the (d) test series are summarized in Table 17.33: For each test run we have collected the models with best fit on training data as well as those that perform best on validation data; average values are given as well as standard deviations. Obviously, rather strong overfitting has happened here; as we had expected, the production of very large formulas leads to over-fit formulas that are not able to perform well on samples that were not used during the training phase.

17.6.3.4 Strong Pruning

Rather strong pruning was applied in test series (e), and as we see in Table 17.34, the formulas produced by GP are significantly smaller than those produced in the previous test series. Still, we observed the fact that genetic diversity is lost very quickly: Already in early stages of the evolutionary processes, the average structural similarity of solutions reaches a very high level (which is documented in the two most right columns of Table 17.34).
The quality of the best models produced is very bad (above 5,000), which is why we do here not state any further details about the evaluation of these models on the given data partitions. We suppose that this low results quality is connected to the loss of population diversity (and of course also the fact that the pruning operations applied were allowed to decrease the models' quality).

Table 17.34: Formula size and population diversity progress in test series (e).

Test series	*Iteration*	*Formula size*		*Solutions similarity*	
		avg	*std*	*avg*	*std*
(1e)	50	12.82	15.76	0.8912	0.0912
	100	18.27	18.15	0.9371	0.0289
	500	19.75	23.52	0.9685	0.0187
	2000	21.39	20.87	0.9891	0.0095
(2e)	10	15.77	9.23	0.9574	0.0318
	20	19.86	10.83	0.9825	0.0247
	50	21.64	16.34	0.9921	0.0082
	End of run	20.03	18.27	0.9943	0.0093

17.6.3.5 Increased Pruning

As light, medium and strong pruning did not lead to the desired results, we have also tried increasing pruning as defined in test strategy (f). As we see in Table 17.35, this strategy performs rather well: The size of the formulas produced by GP rises

especially in early stages of the GP process, but then decreases and on average finally reaches values between 80 and 100.
In addition to this, the population diversity stays higher in the beginning than in GP tests including constantly strong pruning, but eventually decreases and the solutions finally show higher similarities due to the increased pruning in later algorithmic stages.

Table 17.35: Formula size and population diversity progress in test series (f).

Test series	*Iteration*	*Formula size*		*Solutions similarity*	
		avg	*std*	*avg*	*std*
(1f)	50	62.72	95.76	0.3674	0.0943
	100	91.27	130.77	0.3897	0.1059
	500	92.43	107.41	0.6820	0.1124
	2000	87.02	90.68	0.8035	0.0861
(2f)	10	40.78	31.47	0.5235	0.0612
	20	63.59	59.34	0.7052	0.0803
	50	80.26	40.99	0.9450	0.0588
	End of run	79.45	47.67	0.9967	0.0156

The quality values of the results produced in this test series are summarized in Table 17.36. Obviously, less overfitting has happened than in the tests with light or medium pruning.

Table 17.36: Quality of results returned in test series (f).

Test series	*Evaluation data*	*Best model selection basis*			
		Training data		*Validation data*	
		avg	*std*	*avg*	*std*
(1f)	Training data	2,597.35	542.04	7,781.28	827.83
	Validation data	8,904.91	611.02	5,981.52	974.31
	Test data	3,786.51	800.38	2,830.78	427.08
(2f)	Training data	2,275.24	649.11	3,814.93	850.89
	Validation data	9,712.98	767.56	5,862.62	518.53
	Test data	4,912.38	1,198.58	2,275.03	931.62

17.6.3.6 Complexity Dependant Quality Punishment

In fact, our GP test runs including complexity dependant quality punishment, i.e. those of test strategy (g), were also able to produce acceptable results for the NO_x data set investigated here. As we see in Table 17.37, in standard GP the formula sizes are rather high in the beginning and then decrease steadily, whereas in GP with offspring selection the models on average include between 50 and 60 nodes during the whole execution of the GP processes. Population diversity values are comparable to those reported for GP tests without pruning or quality dependant punishment as summarized for example in Section 17.3.
Figure 17.30 illustrates the formula complexity progress of an exemplary GP run of test series (2g).
The qualities of the models with best fit on training and validation are summarized in Table 17.38.

Table 17.37: Formula size and population diversity progress in test series (g).

Test series	*Iteration*	*Formula size*		*Solutions similarity*	
		avg	*std*	*avg*	*std*
(1g)	50	140.76	90.75	0.3824	0.0534
	100	92.62	71.23	0.3916	0.0620
	500	73.73	64.99	0.6381	0.0825
	2000	79.07	47.61	0.7202	0.0696
(2g)	10	50.24	64.67	0.4873	0.0836
	20	60.71	59.01	0.5412	0.0741
	50	65.34	48.33	0.8904	0.0852
	End of run	58.82	41.87	0.9315	0.0423

17.6.3.7 Fixed Size Limits

In the case of fixed size limits the crossover and mutation operators have to consider limits for the complexity of models. Model size and population diversity statistics for test series (h) are summarized in Table 17.39; in GP with offspring selection all formulas eventually are maximally big, and the solutions similarity values show results comparable to those reported in Section 17.3. Table 17.40 summarizes the quality of the results produced, again evaluated on training, validation and test data. Figure 17.31 illustrates the formula complexity progress of exemplary GP test runs of series (1h) and (2h).

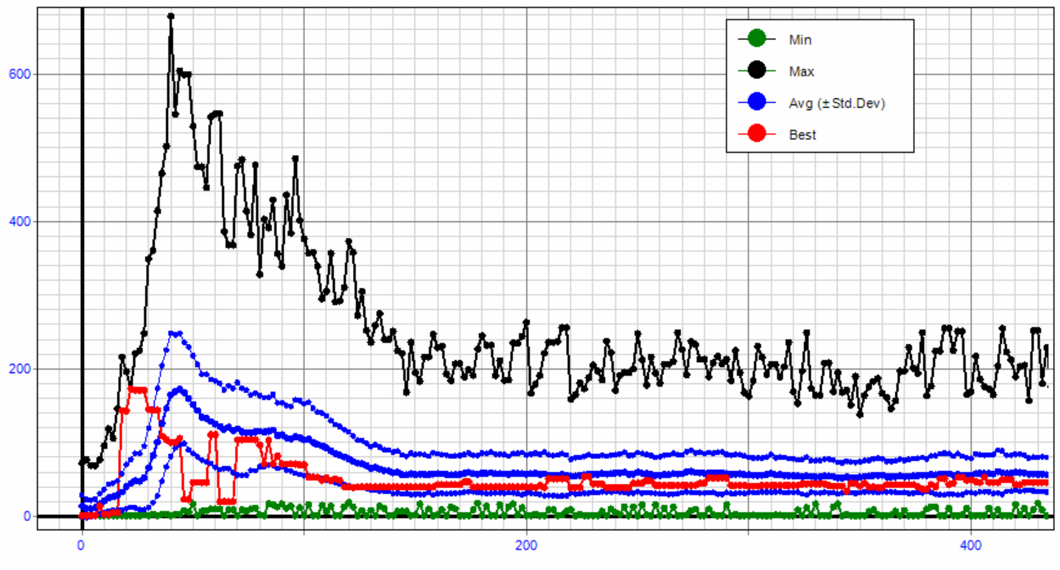


Figure 17.30: Progress of formula complexity in one of the test runs of series (1g), shown for the first ∼400 iterations.

Table 17.38: Quality of results returned in test series (g).

Test series	*Evaluation data*	*Best model selection basis*			
		Training data		*Validation data*	
		avg	*std*	*avg*	*std*
(1g)	Training data	1,837.84	526.10	4,729.42	480.36
	Validation data	12,902.67	767.35	4,531.73	588.30
	Test data	2,597.73	835.41	2,708.36	825.64
(2g)	Training data	1,402.19	593.84	3,121.86	773.91
	Validation data	9,345.87	738.60	3,949.64	962.03
	Test data	2,853.62	812.51	2,618.94	664.07

In addition to total statistics we shall also discuss two selected models returned by one of the test runs of series (2h): Model b_t is the model the performs best on training data (shown in Figure 17.32), b_v the one that performs best on validation data (shown in Figure 17.33). The error distributions on training, validation and test data partitions are illustrated in Figure 17.34.
Table 17.41 characterizes the performance of b_t and b_v by means of mean squared errors as well as the integral values. For this we have calculated the sum of the target values on training, validation and test data and compared these integral values to those calculated using the models under investigation. Obviously, b_t shows a better

Table 17.39: Formula size and population diversity progress in test series (h).

Test series	*Iteration*	*Formula size*		*Solutions similarity*	
		avg	*std*	*avg*	*std*
(1h)	50	37.4182	6.3174	0.4151	0.0935
	100	40.2866	5.8133	0.7231	0.0729
	500	41.7823	4.3973	0.8175	0.0326
	2000	44.2108	5.0450	0.8629	0.0271
(2h)	10	22.4965	8.3763	0.3973	0.0386
	20	27.6203	4.2514	0.6022	0.0493
	50	44.9120	6.4871	0.8907	0.0371
	End of run	50.0000	0.0000	0.9751	0.0189

Table 17.40: Quality of results returned in test series (h).

Test series	*Evaluation data*	*Best model selection basis*			
		Training data		*Validation data*	
		avg	*std*	*avg*	*std*
(1h)	Training data	1,774.94	300.51	4,168.30	1,186.62
	Validation data	10,801.77	923.04	4,248.37	858.02
	Test data	5,791.25	1,266.51	2,610.64	930.44
(2h)	Training data	1,568.12	382.04	3,083.64	502.75
	Validation data	9,641.89	833.71	3,738.13	504.89
	Test data	4,802.30	1,371.22	1,374.61	704.73

integral fit on training (and also validation) data, but when it comes to test data the model that performed best on validation data (b_v) produces much more satisfying results (with an integral error of only 2.354% on test data).

17.6.3.8 Fixed Size Limits and Occasional Pruning

Finally, test series with fixed size limits and occasional pruning have also been executed and analyzed; the results regarding formula complexity, population diversity and results qualities are summarized in Tables 17.42 and 17.43.

Obviously, the results produced are (with respect to evaluation quality) comparable to those produced in the previous series. Still, of course the formula sizes are

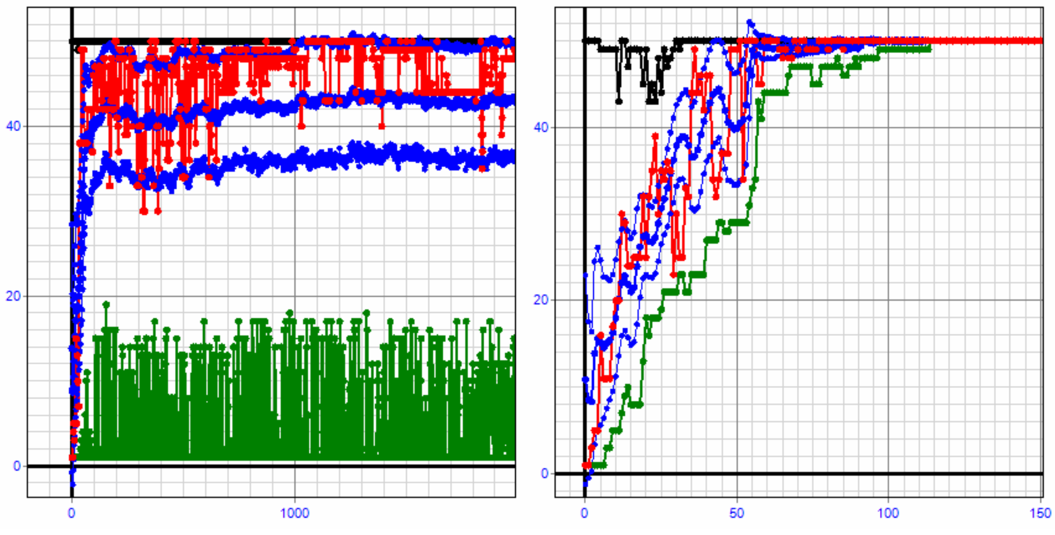
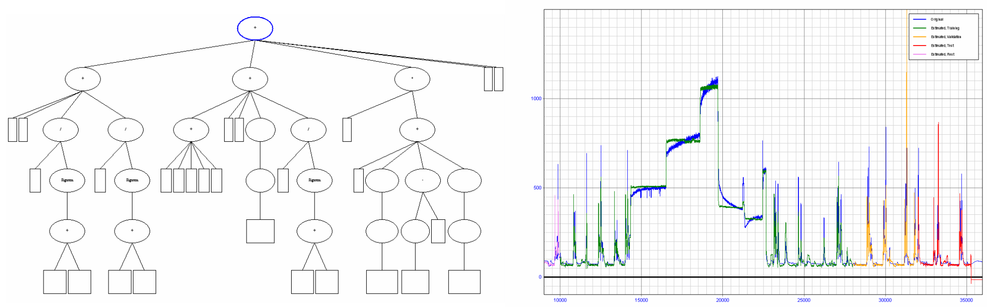


Figure 17.31: Progress of formula complexity in one of the test runs of series (1h) (shown left) and one of series (2h) (shown right).

Table 17.41: Comparison of best models on training and validation data (b_t and b_v, respectively).

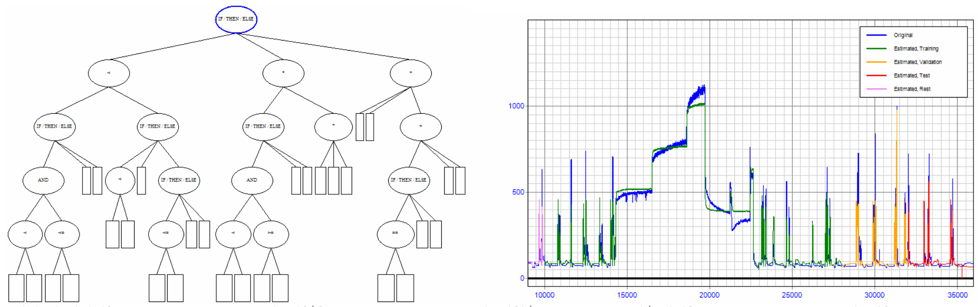
	b_t	b_v
Training quality (MSE)	1,434.65	2,253.62
Validation quality (MSE)	9,187.53	3,748.61
Test quality (MSE)	2,936,40	1,461.95
Target training values integral	$6.010 * 10^6$	
Estimated training values integral	$6.037 * 10^6$ (-0.452%)	$6.084 * 10^6$ (+1.220%)
Target validation values integral	$4.660 * 10^5$	
Estimated validation values integral	$4.620 * 10^5$ (+0.872%)	$4.517 * 10^6$ (+3.173%)
Target test values integral	$3.978 * 10^5$	
Estimated test values integral	$3.198 * 10^5$ (+24.395%)	$3.886 * 10^6$ (+2.354%)

a bit smaller (due to pruning), and also overfitting seems to have decreased: Even though the fit on training data is not as good as on previous test series, the quality on test data is still very good and comparable to the test performance reached in test series (g) and (h).



```
+(+([1,030496*Var003(t-0)]|-([1,02955555783691*Var007(t-8)]|[1,02955555783691*Var007(t-9)])|/([0,859604*Var007(t-7)]|Signum(+(-19,6774250843158
|[0,925277*Var007(t-0)]))))|/([0,796922*Var008(t-10)]|Signum(+(-13,4653753764306|[1,009622*Var007(t-1)]))))|+(+([1,064909*Var006(t-6)]
|[0,245936*Var001(t-5)]|-8,11313346378627|-([1,02466876708512*Var007(t-10)]|[1,02466876708512*Var007(t-11)])|-([1,02993590500011*Var007(t-10)]
|[1,02993590500011*Var007(t-11)]))|[1,062877*Var006(t-6)]|[0,792079*Var008(t-10)]|Exp(Sqrt([1,008566*Var007(t-1)]))|/([0,796922*Var008(t-10)]
|Signum(+(-19,6774250843158|[1,009622*Var007(t-1)]))))|*([1,030496*Var003(t-0)]|+([1,070487*Var007(t-0)]|Signum(Sin([1,015816*Var004(t-6)]))|-
(Signum([0,925277*Var007(t-0)])|[1,028206*Var003(t-7)])|Exp(Sin([0,859604*Var007(t-7)]))))|-([1,02015787437991*Var007(t-9)]
|[1,02015787437991*Var007(t-10)])|-([1,02955555783691*Var007(t-8)]|[1,02955555783691*Var007(t-9)]))
```

Figure 17.32: Model with best fit on training data: Model structure and full evaluation.



```
IF(<(IF(&&(<(-([0,878205630205626*Var009(t-0)]|[0,878205630205626*Var009(t-1)])|14,6416958621683)|<=(-([0,916623127059497*Var002(t-6)]
|[0,916623127059497*Var002(t-7)])|14,6416958621683)))THEN(17,4222949615303),ELSE([1,174105*Var003(t-7)])|IF(<([0,928868*Var008(t-4)]|-
([0,916872632186935*Var006(t-5)]|[0,916872632186935*Var006(t-6)])))THEN([0,970435*Var007(t-9)]),ELSE(IF(<=(-([0,806610292256365*Var002(t-1)]
|[0,806610292256365*Var002(t-2)])|13,4084736247274))THEN([0,974565*Var007(t-0)]),ELSE([1,085533*Var004(t-1)]))))THEN(*(IF(&&(<([0,973703*Var004(t-6)]
|-20)|>=([1,176232*Var003(t-10)]|[0,983657*Var007(t-0)])))THEN(13,4084736247274),ELSE([0,983657*Var007(t-0)])|+([0,796939*Var003(t-0)]
|[1,176232*Var003(t-10)]|13,4084736247274))),ELSE(+(-20|[1,085533*Var004(t-1)]|+(IF(==([0,970435*Var007(t-9)]|-20))THEN([0,786266*Var003(t-10)]),
ELSE([0,983657*Var007(t-0)])|[0,970435*Var007(t-9)]|[0,983657*Var007(t-0)])))
```

Figure 17.33: Model with best fit on validation data: Model structure and full evaluation.

17.6.4 Conclusion

In this section we have demonstrated the effects of code bloat and selected prevention strategies for GP. As expected and known from literature, without any limitations or size reducing strategies GP tends to produce bigger and bigger models that fit the given training data, but of course this also increases the probability of producing over-fit models. Pruning strategies have been analyzed, and the test results show that only strong pruning is able to prevent GP from producing bigger and bigger models, which again decreases population diversity and leads to results which are not optimal. Complexity dependent fitness punishment as well as fixed size limits enable GP to produce quite good results; occasional pruning in combination with fixed size limits can help to decrease overfitting.

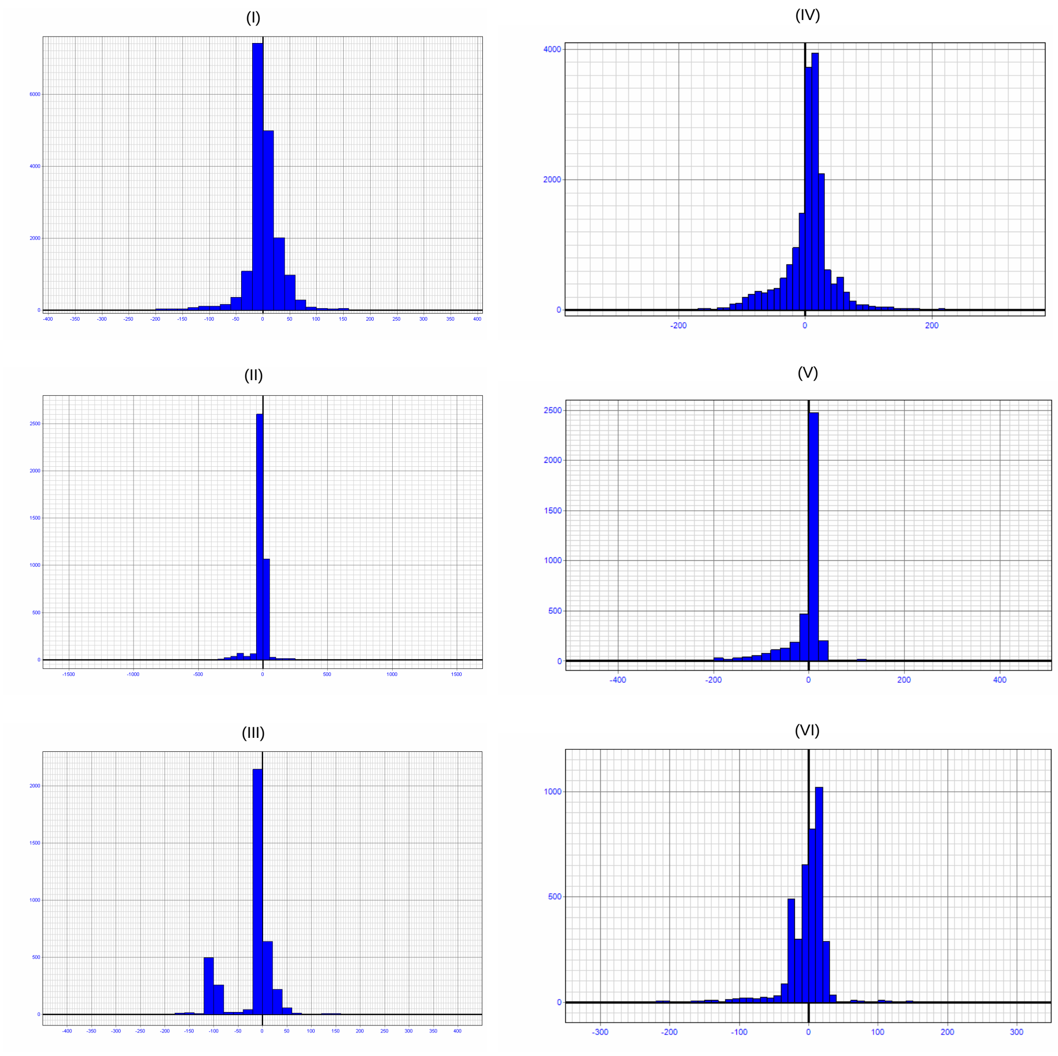


Figure 17.34: Errors distributions of best models: Charts I, II and III show the errors distributions of the model with best fit on training data evaluated on training, validation and test data, respectively; charts IV, V and VI show the errors distributions of the model with best fit on validation data evaluated on training, validation and test data, respectively.

Table 17.42: Formula size and population diversity progress in test series (i).

Test series	*Iteration*	*Formula size*		*Solutions similarity*	
		avg	*std*	*avg*	*std*
(1i)	50	34.8365	6.1534	0.4682	0.0852
	100	37.1863	4.9901	0.7413	0.0711
	500	39.2217	5.2673	0.8388	0.0450
	2000	40.1260	4.9724	0.8992	0.0251
(2i)	10	18.5330	6.6114	0.4307	0.0518
	20	21.5286	5.3083	0.7202	0.0772
	50	38.5143	5.6305	0.9248	0.0403
	End of run	48.2051	4.6228	0.9859	0.0178

Table 17.43: Quality of results returned in test series (i).

Test series	*Evaluation data*	*Best model selection basis*			
		Training data		*Validation data*	
		avg	*std*	*avg*	*std*
(1i)	Training data	2,258.22	561.27	5,869.40	1.233.09
	Validation data	6,608.26	1,463.49	4,819.26	730.51
	Test data	2,238.61	983.57	1,811.05	834.83
(2i)	Training data	1,723.07	623.11	4,209.57	499.89
	Validation data	6,361.46	921.26	3,607.13	736.05
	Test data	3,289.33	945.79	1,434.63	739.22

Chapter 18

Incorporation of A Priori Knowledge: Modeling NO_x with Physical Knowledge and GP

18.1 Physical Knowledge about the Formation of NO_x

In previous chapters we have already described results of attempts to identify dynamic models for NO_x emissions of diesel engines. In fact, research in this area has been done since several decades, so there is already a lot of physical and chemical knowledge available for this modeling task:

As Warnatz, Maas and Dibble explain in [WMD96], the products of combustion are distinctly identified as a severe source of environmental damage, especially caused by increased combustion of hydrocarbon fuels. The major combustion productions, especially carbon dioxide and water, have long been considered rather "harmless"; now, carbon dioxide is more and more seen as a significant source of problems regarding the atmospheric balance and greenhouse effect.

Nitric oxides (NO_x) are less obvious products of combustion; within the last half of the twentieth century it has become apparent that NO and NO_2, collectively called NO_x, are major contributors to photochemical smog and ozone in the troposphere [Sei86]. Gaining knowledge regarding the production process of NO_x is therefore of great interest and researchers search for models for the production of these pollutants in order to find new ways how to minimize them [WMD96].

Based on physical models summarized in [WMD96], we have used the following model describing the production of NO_x dependent on measurable engine parameters. This model has been suggested by Markus Hirsch, research assistant at the Institute for Design and Control of Mechatronical Systems at Johannes Kepler University Linz, Austria [HW07] and can be formulated in the following way:

$$HFM^* = \frac{HFM}{N} \cdot \frac{1000}{60} \left[\frac{kg/h}{U/min} \cdot \frac{1000}{60} = g/U \right] \tag{18.1}$$

$$NO_x \approx e^{(qMI \cdot (\alpha \cdot pMI + \beta \cdot \frac{1}{N} + \gamma \cdot HFM^*))} \tag{18.2}$$

where

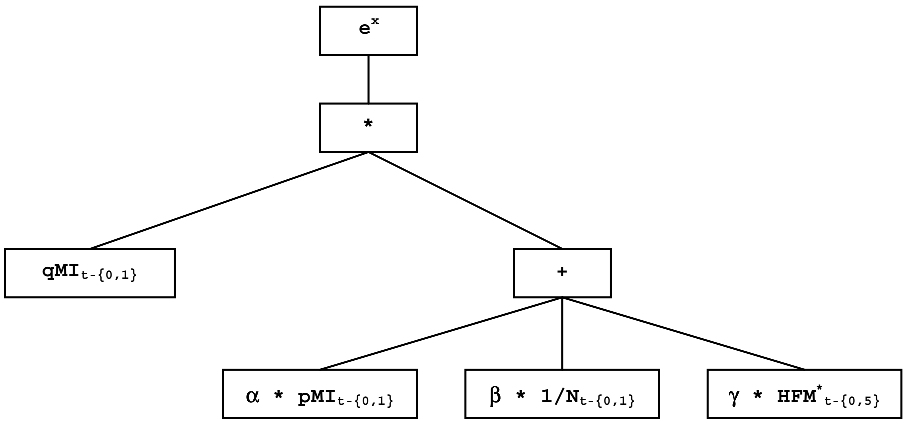
- HFM is the amount of fresh air in the engine's air intake section,
- N the engine's rotational frequency,
- HFM^* is the amount of fresh air divided by the rotational frequency and converted to the amount of air per combustion cycle,
- qMI the amount of fuel injected into the combustion chamber(s),
- pMI is the angle ϕ of the fuel injection, the crankshaft angle[1], and
- α, β and γ are parameters which have to be identified.

Figure 18.1 shows a graphical representation of this semi-abstract model structure available as physical knowledge for the formation of NO_x emissions.

The data set available in this context again contains measurements taken from a 2 liter 4 cylinder BMW diesel engine at a dynamical test bench (simulated vehicle: BMW 320d Sedan). The most recent NO_x data set (III, described in Section 14.2.2) has been used in the test series reported on in this chapter: Several emissions (including NO_x, CO and CO_2) as well as several other engine parameters were recorded at 100 Hz and downsampled to 10 Hz. The maximum time offsets for all potential input variables has been set to 1, only when referencing HFM we have allowed a maximum offset of 5 samples.

Figure 18.2 visualizes the target variable *HoribaNOx* (this figure has on fact already been shown in Section 14.2.2 as Figure 14.8).

[1]The crankshaft angle is directly related to the piston position, which plays an important role in the injection timing. This crankshaft angle can be easily measured and is therefore used for the control of the injection timing.



e^x
*
$qMI_{t-\{0,1\}}$
+
$\alpha * pMI_{t-\{0,1\}}$
$\beta * 1/N_{t-\{0,1\}}$
$\gamma * HFM^*_{t-\{0,5\}}$

Figure 18.1: Model representing the physical knowledge available for the formation of NO_x emissions.

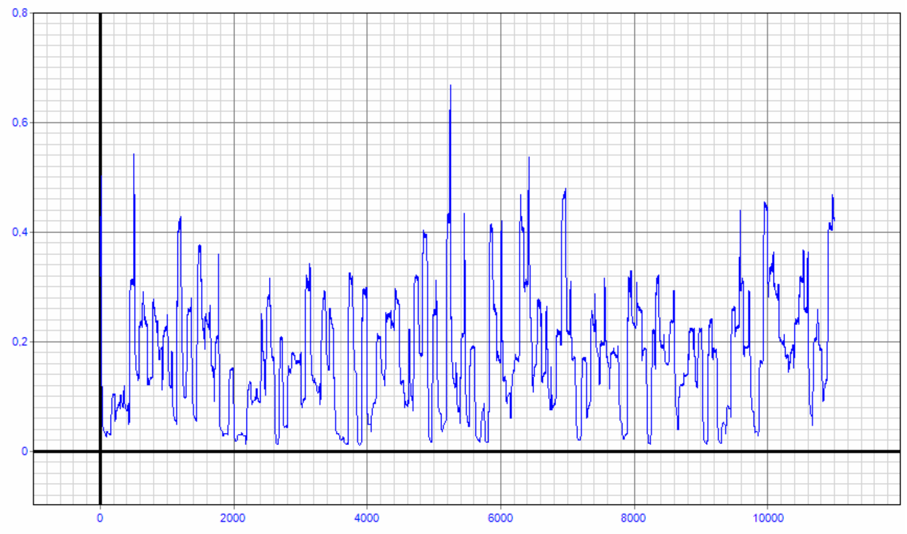


Figure 18.2: Target *HoribaNOx* values of NO_x data set III.

18.2 Strategies for the Incorporation of Knowledge about the Formation of NO_x in GP

As we now know about the physical knowledge available in context with formation of NO_x during combustion in diesel engines, there are several ways how we can make this information available for the GP process.

18.2.1 Introduction of a New Variable for HFM*

First and most obviously, Formula 18.1 describing the calculation of the auxiliary variable HFM^* can be used for defining a new variable; as there are no parameters to be fixed, this new variable can be introduced into the data base immediately. The GP process is therefore able to use this information simply by using this new variable, i.e. by creating models that reference the variable HFM^*.

18.2.2 Seeding Stub Models for NO$_x$

The incorporation of the information given in Formula 18.2 is a bit more complicated as it includes parameters which are not known; we shall here discuss possibilities how to use the strategies given in Chapter 9.

The first possibility is to seed the population using a stub of the model already known. Of course, this brings along the problem that the unknown parameters included in the model, namely α, β and γ, have to be initialized using some arbitrary, but fixed values; we initially set those parameters to 0.1 and expect the evolutionary optimization process to tune the values so that improved model structures are evolved.

As already mentioned in Chapter 9, we now have to decide to which extent this pre-defined model shall be inserted in the initial population and during the main loop of the GP process.

18.2.3 Defining Terminals and a Basic Function for NO$_x$

An alternative method is to create an artificial function that represents the structure of the knowledge available. So we define the following additional items that are to be added to the functional basis used by the GP process:

- The function definition "PKfuncNOx" represents the main part of the model. On the one hand it expects the input variables qMI, pMI, $1/N$ (N^{-1}) and HFM^* as inputs at indices 0, 2, 4 and 6; on the other hand it also expects 5 more inputs that are processed as coefficients (at indices 1, 3, 5 and 8) or an additional term at index 7.
  When called with the expected inputs $I_{0...8}$, this function returns the result of

the following expression:

$$e^{I_0 \cdot (I_1 \cdot I_2 + I_3 \cdot I_4 + I_5 \cdot I_6 + I_7) \cdot I_8} \tag{18.3}$$

- The terminal definitions "qMI", "pMI" and "HFM*" represent variables that always reference the variables qMI, pMI and HFM^*, respectively, and
- the terminal definition "Ninv", in Figure 18.3 shown as "N^{-1}", always returns the multiplicative inverse of the respective sample in the N variable.

The requirements regarding valid parent and child definitions are given in such a way that a node referencing the "PKfuncNOx" function is only allowed to accept child nodes referencing the terminal definitions "qMI", "phiMI", "N^{-1}" and "HFM*" at indices 0, 2, 4 and 6, respectively. All other indices are not restricted, i.e. any subtree structure can be attached at indices 1, 3, 5, 7 and 8 representing the parameters α, β and γ, an additional additive term and an additional coefficient.

Figure 18.3 graphically shows the terminals and the function definition used in this approach.

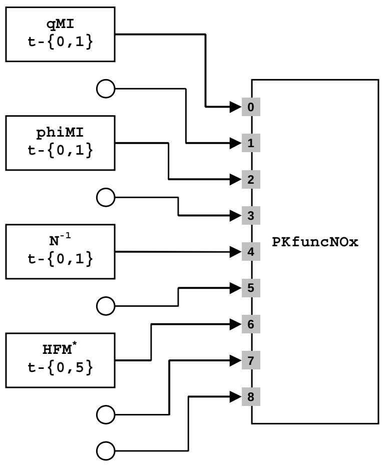


Figure 18.3: Terminal definitions and the "PKfuncNOx" function representing physical knowledge about the formation of NO_x emissions.

18.3 Test Strategies

Using the definitions described in the previous section we shall now see to which extent these strategies are appropriate for incorporating physical knowledge into the GP process. We here report on test series executed using the test strategies summarized in Table 18.1.

Table 18.1: Test strategies for incorporating physical knowledge about the formation of NO_x in the GP process.

Strategy Index	*Approach and Parameters*
(I)	No additional information given.
(II)	Use of additional variable HFM^*, as described in Section 18.2.1.
(III)	Use of additional variable HFM^* as well as stub model described in Section 18.2.2.
(IIIa)	Seed model in 20% of the initial population
(IIIb)	Seed model in 60% of the initial population
(IIIc)	Seed model in 100% of the initial population
(IV)	Use of additional variable HFM^* as well as stub model described in Section 18.2.2.
(IVa)	Seed model in 10% of the initial population and with 10% probability in main loop (replacing solutions selected by parents selection)
(IVb)	Seed model in 20% of the initial population and with 20% probability in main loop (replacing solutions selected by parents selection)
(IVc)	Seed model in 50% of the initial population and with 30% probability in main loop (replacing solutions selected by parents selection)
(V)	Use of additional variable HFM^* as well as terminals and the "PKfuncNOx" function as described in Section 18.2.3
(Va)	No manipulation of the GP-process
(Vb)	Introduction of "PKfuncNOx" function into solutions by special mutation operator; probability: 15%

Once again, all test strategies have been executed 5 times independently: We applied GP with 1000 solutions, strict offspring selection (maximum selection pressure: 200), gender specific parents selection (random & roulette) and 12% mutation rate.

18.4 Test Results

18.4.1 Test Series I: Using no Additional Information

Table 18.2 summarizes the quality of the models produced for the NO_x data set without adding any physical knowledge. We here also give average and standard deviation values of the performance of the models that show best fit on training and of those that show best fit on validation data. Once again we see that those models, that perform best on validation data, are also those that perform quite well on test data (at least better than those that show best fit on training data).

Table 18.2: Quality of results produced in test series I.

Model	*Quality (mse)*					
	Training		*Validation*		*Test*	
	avg	*std*	*avg*	*std*	*avg*	*std*
Best on training data	0.003585	0.000241	0.004497	0.000341	0.005236	0.000382
Best on validation data	0.004148	0.000276	0.003538	0.000295	0.004359	0.000331

18.4.2 Test Series II: Using an Additional Variable

After adding physical knowledge in terms of the additional variable HFM^* we again executed GP tests and collected the results summarized in Table 18.3. Obviously, the results are a lot better - not only in terms of training, but also validation and test quality.

Table 18.3: Quality of results produced in test series II.

Model	*Quality (mse)*					
	Training		*Validation*		*Test*	
	avg	*std*	*avg*	*std*	*avg*	*std*
Best on training data	0.003312	0.000219	0.003714	0.000204	0.003979	0.000263
Best on validation data	0.004022	0.000157	0.003329	0.000198	0.003901	0.000374

18.4.3 Test Series III and IV: Inducing Model Structures into GP

In addition to the variable HFM^*, physical knowledge has also been inducted into the GP processes by seeding known model structures into the initial population as well as into the main GP loop as defined in test strategies III and IV; Tables 18.4 and 18.5 summarize the quality of the models produced in these test series. Obviously, this induction of model structures again brought along an increase of the results' quality:

- Seeding the models into the initial population at medium rate (series IIIb) seems to be the best strategy if no more manipulation is done during the main GP loop.
- Regarding the induction of models in the main GP loop (as done in test series IV) also the medium variant seems to have performed best, as the results of series IVb are better than those measured for test series IVa and IVc.

Table 18.4: Quality of results produced in test series III.

Model		*Quality (mse)*					
		Training		*Validation*		*Test*	
		avg	*std*	*avg*	*std*	*avg*	*std*
(a)	*Best on training data*	0.00327	0.00018	0.00359	0.00019	0.00391	0.00018
	Best on validation data	0.00390	0.00015	0.00316	0.00010	0.00388	0.00022
(b)	*Best on training data*	0.00319	0.00020	0.00381	0.00022	0.00382	0.00031
	Best on validation data	0.00404	0.00014	0.00305	0.00014	0.00408	0.00038
(c)	*Best on training data*	0.00361	0.00025	0.00393	0.00028	0.00409	0.00024
	Best on validation data	0.00463	0.00026	0.00329	0.00017	0.00429	0.00032

Figure 18.4 illustrates a model which was returned as best model with respect to fit on validation data in one of the test runs in series IVb. Obviously, the given model structures that were inducted into the GP processes in series IV have been used and are incorporated in the model's structure.

Of course, all these manipulations of the GP process (as done in test series III and especially IV) are expected to have more or less significant effects on the GP population dynamics. This can be clearly seen in Tables 18.6 and 18.7 which summarize the average solution similarities (given as mean average and standard

Table 18.5: Quality of results produced in test series IV.

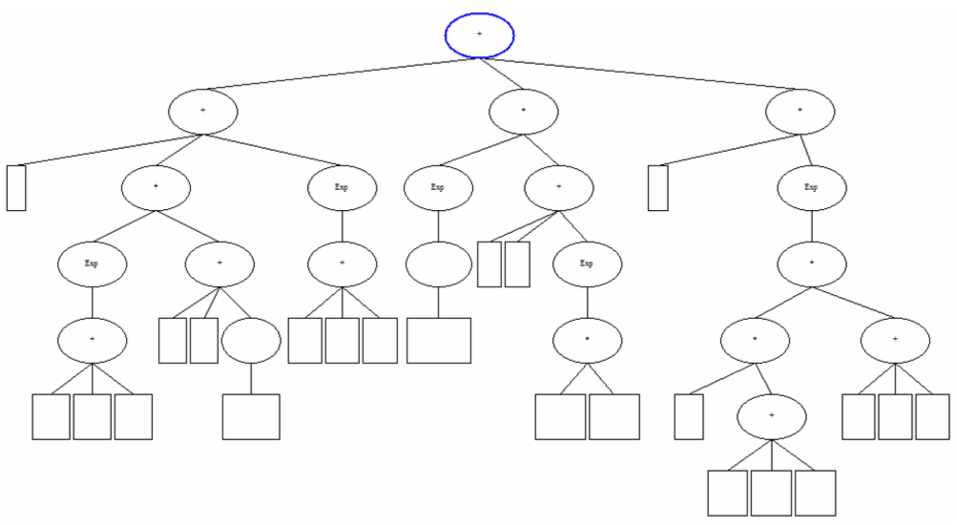
Model		*Quality (mse)*					
		Training		*Validation*		*Test*	
		avg	*std*	*avg*	*std*	*avg*	*std*
(a)	*Best on training data*	0.00316	0.00016	0.00382	0.00017	0.00402	0.00023
	Best on validation data	0.00399	0.00016	0.00339	0.00014	0.00382	0.00032
(b)	*Best on training data*	0.00289	0.00015	0.00360	0.00017	0.00370	0.00017
	Best on validation data	0.00439	0.00015	0.00289	0.00010	0.00348	0.00020
(c)	*Best on training data*	0.00305	0.00027	0.00427	0.00024	0.00426	0.00026
	Best on validation data	0.00387	0.00020	0.00353	0.00019	0.00384	0.00027

deviation values) of the test series III and IV: Of course, the more the GP process is forced to use some given model structure (parts), the less the population diversity becomes and more the mutual similarity of the populations' individuals rises. This effect becomes obvious especially in the results for test series IIIc and IVc.

Table 18.6: Population diversity progress in test series III.

Test series	*Iteration*	*Average solutions similarity*	
		avg	*std*
a	10	0.2738	0.0398
	20	0.4380	0.0479
	40	0.7429	0.0334
	End of run	0.9892	0.0247
b	10	0.3952	0.0402
	20	0.4537	0.0537
	40	0.7548	0.0479
	End of run	0.9917	0.0281
c	10	0.4203	0.0458
	20	0.4946	0.0435
	40	0.7823	0.0382
	End of run	0.9946	0.0172

Furthermore, forcing the GP process to use some given model structures also influences population dynamics not only in terms of population diversity, but also regarding the frequency and the impact of terminals and functions that are available in the given functional basis. This can be done measuring mean squared difference function defined in Equation 12.19; terminals and functions are systematically removed (i.e., replaced by their parents' neutral elements for the respective input index) and the formulas re-evaluated. Table 18.8 summarizes the average fitness-weighted impact values for each given terminal and variable in test series I & II and



```
*(0,03295252415|+(+(+(-9,46567338989098|*(Exp(+(*(2,28775692582664|+([0,070808*Var008(t-0)]|-0,258062238956865))|*(0,0332778702163062|+([-0,076378
*Var012(t-2)]|168,527475406471))|*(13,1447989812029|+([0,134846*Var024(t-0)]|-0,116801040367473))))|+(*(0,0216216216216216|+([-0,233247*Var013(t-0)]
|336,692581340245))|*(13,1447989812029|+([0,220482*Var024(t-0)]|-0,190977552643847))|Exp(*(8,06451612903226|+([-0,053086*Var009(t-2)]
|0,0658266640240317)))))|Exp(+(*(13,1447989812029|+([0,072412*Var024(t-0)]|-0,0627215386548598))|*(0,0216216216216216|+([-0,042651*Var013(t-0)]
|61,5662161068429))|*(13,1447989812029|+([0,154444*Var024(t-0)]|-0,133776754704841)))))|*(Exp(Cos(*(13,1447989812029|+([0,167279*Var024(t-0)]|-
0,144893638173656))))|+(*(0,088308019927729|+([0,128883*Var014(t-2)]|-16,5470311567856))|*(0,0332778702163062|+([-0,022459*Var012(t-2)]
|49,5555169769855))|Exp(*(*(0,088308019927729|+([-0,146616*Var014(t-1)]|18,8237422359773))|*(13,1447989812029|+([0,165896*Var024(t-0)]|-
0,143696015960345))))))|*(*(1,43369187960074|+([-0,101744*Var007(t-2)]|1,52463483246412))|Exp(*(*(*(1,43369187960074|+([-0,008179*Var007(t-0)]
|0,122568034142857))|+(*(2,28775692582664|+([-0,088277*Var008(t-1)]|0,321729996215363))|*(13,1447989812029|+([0,157689*Var024(t-0)]|-
0,136587034346519))|-0,463536465954036))|+(*(0,088308019927729|+([0,181987*Var014(t-1)]|-23,3649753346804))|-9,8625925134551
|*(0,0455062540041238|+([0,079715*Var011(t-1)]|-56,4100106106371)))))))|10,3291258342041))
```

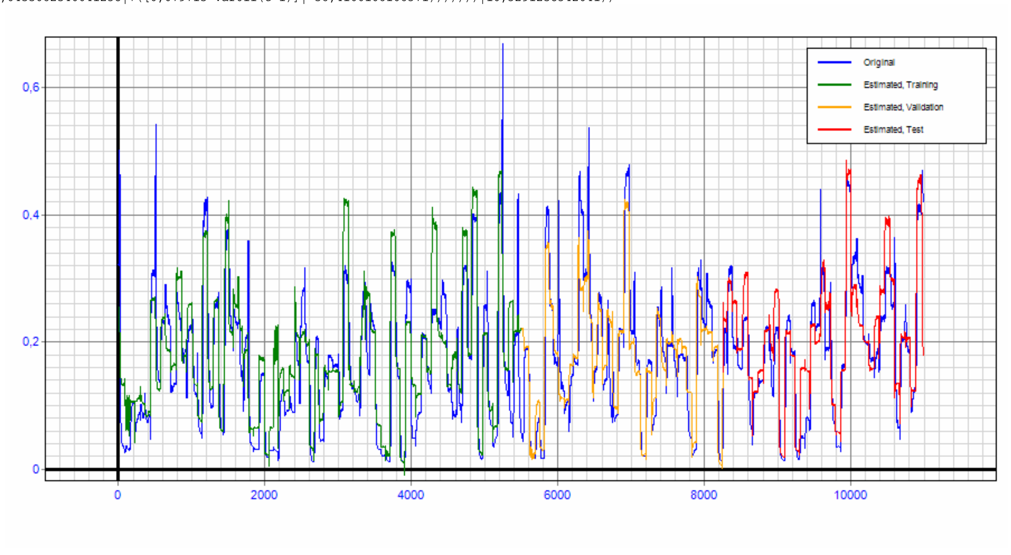


Figure 18.4: Best model produced for the NO_x data set: The given a priori knowledge has been incorporated as subtrees of the returned model structure.

III & IV[2] after 10 iterations and at the end of the GP runs that have been analyzed. As we see here, some terminals and variables tend to have almost no impact on the populations' evaluation (especially differential, square root, logarithm, sine, condi-

[2]For the sake of results lucidity the results for test series I and II have been collected and analyzed collectively as well as those for test series III and IV.

Table 18.7: Population diversity progress in test series IV.

Test series	*Iteration*	*Average solutions similarity*	
		avg	*std*
a	10	0.2664	0.0428
	20	0.4334	0.0476
	40	0.7796	0.0309
	End of run	0.9914	0.0187
b	10	0.2703	0.0438
	20	0.4737	0.0385
	40	0.8290	0.0321
	End of run	0.9953	0.0196
c	10	0.3025	0.0422
	20	0.5916	0.0429
	40	0.8645	0.0313
	End of run	0.9971	0.0127

tionals, and Boolean and logical functions), whereas others are integral parts of the evaluation of the produced GP populations. Furthermore, in the results calculated for test series III and IV we see a higher impact of the multiplication and exponential functions; due to the repeated induction of model structures the functions used in these structures eventually become more prominent, integral parts of the populations' solutions.

18.4.4 Test Series V: Using an Enhanced Functional Basis

Test series V finally represents GP tests using the artificial function "PKfuncNOx" as described in Section 18.2.3. Both surprisingly and also a little disappointingly, this function was not able to enable GP to produce better models; in fact, in both test variants (Va and Vb) this function was not considered by the GP process. Table 18.9 summarizes the quality of the best models produced in these test series: The quality of the models produced in series Va is comparable to the fitness of those of series II (without any manipulation of the GP process), forcing GP to use "PKfuncNOx" by directed mutation here even leads to worse results.

In order to find out whether or not this is only because of the enhanced selection concepts used we have here also tried standard genetic programming (SGP) with 1000 individuals, 7% mutation rate, tournament selection (k=3) and 2000 generations. The quality of the models returned by SGP are (as expected) slightly worse – this can be seen in Table 18.10. Thus, offspring alone cannot be considered to be

Table 18.8: Population diversity progress in test series IV.

Definition	*Test series I and II*		*Test series III and IV*	
	Iteration 10	*End of run*	*Iteration 10*	*End of run*
Variable	23.63	50.16	37.94	54.82
Constant	17.99	28.82	36.85	21.39
Differential	2.03	0.37	3.86	0.68
Addition	31.89	49.27	25.31	31.97
Subtraction	15.79	28.29	13.37	9.62
Division	9.04	7.53	8.20	6.02
Multiplication	59.43	51.16	77.04	80.69
Power	33.37	3.52	28.31	1.35
Square Root	0.01	0.02	0.03	0.01
Exponential	18.36	19.28	14.07	38.23
Logarithm	0.00	0.00	0.00	0.00
Trigonometrics	23.19	9.42	15.68	2.52
Signum	8.29	0.05	9.70	0.00
Boolean	13.03	0.04	9.42	0.00
Logical	8.02	0.01	0.01	0.01
Conditional	13.09	0.85	3.69	0.02

Table 18.9: Quality of results produced in test series V.

Model		*Quality (MSE)*					
		Training		*Validation*		*Test*	
		avg	*std*	*avg*	*std*	*avg*	*std*
(a)	*Best on training data*	0.00341	0.00020	0.00388	0.00021	0.00413	0.00027
	Best on validation data	0.00384	0.00017	0.00334	0.00018	0.00409	0.00032
(b)	*Best on training data*	0.00400	0.00031	0.00445	0.00034	0.00436	0.00044
	Best on validation data	0.00426	0.00017	0.00356	0.00026	0.00388	0.00028

the cause for these results.
We have analyzed the population diversity for test series Va and Vb, both with standard as well as enhanced GP (with gender specific parents selection and strict offspring selection), the results are summarized in Table 18.11: For series Va the results are comparable to those reported on in Section 17.3, the analysis of test series Vb shows higher mutual solutions similarity values and thus lower population diversity.

Of course we have also calculated the impact of the "PKfuncNOx" function[3] in

[3]The impact of the "PKfuncNOx" function is here calculated in the same way as in previous terminals and variables impact comparisons as defined in Section 18.4.3.

Table 18.10: Quality of results produced by standard GP in test series V.

Model		*Quality (MSE)*					
		Training		*Validation*		*Test*	
		avg	*std*	*avg*	*std*	*avg*	*std*
(a)	*Best on training data*	0.00372	0.00031	0.00391	0.00027	0.00549	0.00041
	Best on validation data	0.00419	0.00020	0.00357	0.00023	0.00486	0.00040
(b)	*Best on training data*	0.00387	0.00041	0.00458	0.00040	0.00473	0.00038
	Best on validation data	0.00528	0.00021	0.00386	0.00037	0.00431	0.00031

Table 18.11: Population diversity progress in test series V.

Test series	*Iteration*	*Average solutions similarity*			
		Standard GP		Extended GP	
		avg	*std*	*avg*	*std*
a	10	0.2948	0.0562	0.3520	0.0398
	400 / 20	0.7835	0.0724	0.4251	0.0432
	1000 / 40	0.8620	0.0529	0.6368	0.0379
	End of run	0.8873	0.0235	0.9641	0.0261
b	10	0.3523	0.0511	0.3532	0.0430
	400 / 20	0.8407	0.0604	0.5286	0.0402
	1000 / 40	0.8825	0.0516	0.7753	0.0317
	End of run	0.8933	0.0362	0.9751	0.0185

standard as well as in extended GP:

- In test series Va, the impact of "PKfuncNOx" is on average 21.45 in standard GP after 100 iterations and eventually falls down to 11.89; in extended GP, it reaches 18.52 after 10 iterations and in the end goes down to only 7.41.
- In test series Vb (including repeated introduction of the "PKfuncNOx" function by directed mutation) the impact of this function is of course higher. In standard GP it reaches 28.11 on average after 100 generations and 31.23 by the end of the GP processes; in extended GP, the function's average impact is about 24.58 after 10 generations and finally reaches 35.59.

Thus, comparing these results we have to admit that the "PKfuncNOx" function loses importance almost completely during the GP process executions; only by introducing it repeatedly it can remain in the GP populations, but this again leads to significantly worse results than without doing so.

Obviously, in this application it has been easier for the GP process to integrate physical knowledge given as model structures (built up of simple standard components) than complex functions. We have also observed the fact that the "PKfuncNOx" function was represented in successful solutions only if all input subtrees were terminal nodes - as soon as more complex subtrees were attached to this function (serving as coefficients for the terms that are summed up), the function contributed in a negative way so that the formulas were either eliminated by offspring selection or not selected as parents for the next generations.

18.5 Conclusion

In this chapter we have summarized test results for an example for the introduction of a priori about the system which is to be identified: Virtual sensors for the NO_x emissions of a BMW diesel engine have been created using physical knowledge. Three different ways how to introduce additional knowledge into the GP based learning process have been discussed and tested:

- If partial knowledge can be formulated by equations without variable parameters, then additional variables can be formed. Of course, this approach can be used in any machine learning approach; in this example application, the virtual variable HFM^* has been formed and added to the problem data set, leading to increased model qualities.
- Alternatively, model structures representing partial knowledge (about physical systems, e.g.) can also be introduced into the GP process by seeding parts of the initial population or by repeatedly inserting them into the main GP loop (before crossover and mutation operations). In our example application, this was also successfully done; of course, if this forceful introduction is done too often, then population diversity can be lost leading to worse results.
- The third possibility discussed and tested here is the formation of complex functions representing partial knowledge; the genetic process is then supposed to form models that include these functions. Unfortunately, exactly this approach did not really work fine in this exemplary application: On the one hand, without manipulating the GP process, the function designed in this example died off almost completely, and on the other hand forceful introduction of this function into the existing models had negative effects on population diversity as well as on results quality.

Chapter 19

Results Stability

19.1 Introduction

Even though GP is widely considered a method that is able to identify models for several kinds of linear as well as nonlinear systems (see examples for example in [Koz92], [KKS$^+$03a], [WEA$^+$06] or [WAW07a]), the following problem is repeatedly discussed: Due to the stochastic element that is intrinsic to any evolutionary process and the extremely wide range of possible (arbitrarily complex) models that can be formed using function and terminal definitions, genetic programming cannot guarantee that the results of GP processes applied to a given data set will always be similar or even equal to each other. Still, if there is a physical model underlying to the data that are analyzed, then GP is expected to find these structures and produce somehow similar results.
In principle, the analysis of GP results reported on in this chapter was done in the following way: For each test run (including 10 separate GP processes) we collect the models that were eventually returned by the modeling algorithms; all returned models are then syntactically compared to each other using the multiplicative structural comparison function presented in Chapter 11.

An explanatory, synthetic example is given in Figure 19.1: 5 solutions are compared to each other and the results are summarized in the given table. Additionally, for each solution we collect the maximum similarity of the respective solution compared to all other ones: $maxSim(m_i) = max_{\forall j:1 \leq j \leq n, j \neq i}(sim(m_i, m_j))$ (where m_i is the model number i and n the total number of models).
In this example, the mean similarity is 0.313 (standard deviation: 0.245), and the mean maximum similarity is computed as 0.529 (standard deviation: 0.248).

```
  Iteration 17

    01: Signum(Sqrt(Log(-(e^(4,92436805485784|e^([1,115452*Var029(t-0)]|
          Cos([1,058453*Var012(t-0)])))|*(Signum(-8,98619052616505)|/(Tan(-
          ([0,918539224493133*Var028(t-0)]|[0,918539224493133*Var028(t-1)]))
          |[0,989673*Var029(t-0)])|*(5,50294049733634|[0,990563*Var010(t-0)]))))))
    02: /(Tan(IF(<=(+([1,077744*Var022(t-0)]|-([1,10727685164315*Var015(t-0)]
          |[1,10727685164315*Var015(t-1)]))|-([0,976768789758407*Var022(t-0)]
          |[0,976768789758407*Var022(t-1)])))THEN([1,041900*Var004(t-0)]),
          ELSE(Sqrt(IF(<=(2,44583836239596|[1,154666*Var009(t-0)]))
          THEN([0,943343*Var029(t-0)]),ELSE([0,950025*Var022(t-0)])))))|
          Tan(IF(<=(+([1,077744*Var022(t-0)]|17,1775511725676)|-
          ([0,976768789758407*Var022(t-0)]|[0,976768789758407*Var022(t-1)])))
          THEN(0,908943850130704),ELSE(Sqrt(IF(<=(-1,91706125250988| [1,040191*
          Var027(t-0)]))THEN([0,943343*Var029(t-0)]),ELSE([0,950025*Var022(t-0)])))))
    03: /([1,018166*Var029(t-0)]|[1,018166*Var029(t-0)])
    04: IF(||(||(<=([0,963652*Var028(t-0)]|[1,043041*Var005(t-0)])| >([0,853987*Var027
          (t-0)]|-([1,03592076887582*Var005(t-0)]| [1,03592076887582*Var005(t-1)])))|<(-
          (e^([0,935266*Var000(t-0)]|18,5846317477213)|0,628437844963067)|
          +(18,9154051224144|11,6099712533028)))) THEN(Signum(Signum(Tan(IF(<=(-
          ([1,0425793099*Var005(t-0)]| [1,0425793099*Var005(t-1)])|[1,171622*
          Var013(t-0)]))THEN([0,968277*Var029(t-0)]) ,ELSE(Signum(-(13,4731341989302|-
          2,3347292239722))))))), ELSE(Tan(Signum(+([1,172564*Var013(t-0)]|
          Signum(6,10413791798622)|-([0,959094579076586*Var001(t-0)]|
          [0,959094579076586*Var001(t-1)])))))
    05: IF(&&(&&(<=(-([1,04831623404712*Var023(t-0)]|[1,04831623404712*Var023(t-1)])|
          Log([1,023825*Var001(t-0)]))|>=(-14,6562330699379|-12,6473184478614))|~(<(-
          ([0,932356724857449*Var024(t-0)]|[0,932356724857449*Var024(t-1)])|
          Log(Signum([0,941442*Var004(t-0)]))))))THEN([1,232069*Var001(t-0)]),
          ELSE(Signum(Sin([1,068600*Var029(t-0)]))))

 Similarity |    (01)    |    (02)    |    (03)    |    (04)    |    (05)    |  maxSim
------------+------------+------------+------------+------------+------------+-----------
      (01)  |      -     |  0,143795  |  0,043997  |  0,275474  |  0,140406  |  0,275474
      (02)  |  0,210052  |      -     |  0,072045  |  0,513435  |  0,331605  |  0,513435
      (03)  |  0,939894  |  0,800255  |      -     |  0,379430  |  0,379209  |  0,939894
      (04)  |  0,257190  |  0,337566  |  0,024343  |      -     |  0,419353  |  0,419353
      (05)  |  0,165161  |  0,251798  |  0,032487  |  0,545353  |      -     |  0,545353

mean(similarity):   0,31314
stddev(similarity): 0,24465

mean(maxSim):       0,53870
stddev(maxSim):     0,24757
```

Figure 19.1: Synthetic example for results stability analysis.

The results presented in this chapter are partially to be published in [WAW08b] on the reliability of nonlinear modeling using enhanced genetic programming techniques.

19.2 Test Setup

We have tested GP with strict offspring selection (comparison factor and success ratio have been set to 1.0, the maximum selection pressure to 200) using the following 4 test data sets:

- The *Melanoma* data set,

- the *Thyroid* data set,
- the *Wisconsin* data set, and
- the NO_x data set described in Section 14.2.2.

In all four cases the first 80% of the data sets were used as training data, the rest as validation data (used by pruning methods).

The functions and terminals basis contained the following definitions:

- Basic arithmetic functions (addition, subtraction, multiplication, division),
- exponential and logarithm functions,
- trigonometric functions (sine, cosine and tangent), and
- conditional, boolean and logic functions;
- variables (in the case of learning models for the classification sets no time offsets other than 0 were allowed, for the NO_x data set the maximum time offset was set to 2.0),
- constants, and
- differentials (only in the context of learning models for the NO_x data set).

For calculating the similarity of models the multiplicative structural similarity function has been used (all coefficients $c_1 \dots c_{10}$ were set to 0.2, only the coefficient c_1 weighting the level difference contribution d_1 was set to 0.8.). For each test run we give the average and standard deviation values of the similarities of the models; additionally, for each model we have collected the maximum similarity values *maxSim* and calculated average and standard deviation for these values, too.

19.3 Test Results

The Tables 19.1 – 19.4 summarize the similarity values calculated for each of the 5 test series (for learning models for each data set); the average similarity values and average maximum similarity values are given as well as the respective standard deviations.

Obviously, the results produced are not extremely similar to each other: Given the fact that 1,000 randomly generated models for the NO_x data set show an average similarity of 0.1287 (std : 0.0255), an average similarity of 0.4149 seems plausible, but unfortunately not really high; in fact, we had expected higher values. For the classifiers produced we see even lower results, having in mind that 1,000 randomly generated classifiers for the *Melanoma*, *Thyroid* or the *Wisconsin* data set show average similarities of 0.1478 (std : 0.02736).

Of course, one fact has to be considered here: GP tends to produce bloat, and models produced by GP are likely to become maximally big and are thus prone to overfitting. This is why for example pruning methods are used for removing parts of the models produced in order to become smaller, more compact formulas; still, this pruning should not decrease the quality of the models more than necessary (this trade-off always has to be kept in mind when it comes to pruning). We have used a pruning method based on evolution strategies (as described in Section 10.2) for reducing the complexity of the models that are eventually returned as results of the GP processes by cutting out nodes and deleting subtrees of the given model structures; no quality deterioration was allowed, the number of solutions (λ) was set to 25, and the maximum number of iterations to 10. After doing so, the pruned results were once again compared to each other leading to the results summarized in Tables 19.5 – 19.8.

Obviously, the mutual similarity of the pruned results is a lot higher than the original ones, on average even reaching 0.6238 for the NO_x data set. This result is in fact very satisfying because it shows that independent GP processes have in this case really produced formulas with rather similar model structures; pruning reveals the really essential parts of the models, and these essential parts are more similar to each other than the original ones. Furthermore, this increase of results similarity after pruning them is a lot stronger when analyzing the results for the NO_x data set (representing a real, mechatronical system) than when analyzing the classifiers produced for the classification data sets.

Table 19.1: Results similarity statistics for tests using the *Melanoma* data set.

Test run	*mean(sim)*	*stddev(sim)*	*mean(maxSim)*	*stddev(maxSim)*
1	0.4955	0.1808	0.6478	0.0482
2	0.5115	0.1917	0.6053	0.1352
3	0.4615	0.1511	0.5813	0.0982
4	0.4855	0.2309	0.6310	0.1073
5	0.5198	0.1652	0.6335	0.1130
avg.	*0.4948*	*0.1839*	*0.6198*	*0.1004*

Table 19.2: Results similarity statistics for tests using the *Thyroid* data set.

Test run	*mean(sim)*	*stddev(sim)*	*mean(maxSim)*	*stddev(maxSim)*
1	0.4650	0.1065	0.5918	0.0616
2	0.4852	0.1129	0.6591	0.0475
3	0.4640	0.1026	0.5875	0.0673
4	0.4933	0.1029	0.6053	0.0581
5	0.4472	0.1747	0.6363	0.0973
avg.	*0.4709*	*0.1199*	*0,6160*	*0,0664*

Table 19.3: Results similarity statistics for tests using the *Wisconsin* data set.

Test run	*mean(sim)*	*stddev(sim)*	*mean(maxSim)*	*stddev(maxSim)*
1	0.4438	0.1513	0.6050	0.1229
2	0.4611	0.2040	0.5946	0.1681
3	0.4081	0.1550	0.7164	0.1329
4	0.4252	0.3342	0.7367	0.1833
5	0.4771	0.1459	0.5904	0.1199
avg.	*0.4431*	*0.1981*	*0.6486*	*0.1454*

Table 19.4: Results similarity statistics for tests using the NO_x data set.

Test run	*mean(sim)*	*stddev(sim)*	*mean(maxSim)*	*stddev(maxSim)*
1	0.3991	0.1705	0.6038	0.1673
2	0.4279	0.1546	0.6518	0.0825
3	0.3966	0.1359	0.5761	0.1608
4	0.4009	0.1299	0.6780	0.1265
5	0.4498	0.1300	0.5555	0.0904
avg.	*0.4149*	*0.1442*	*0.6130*	*0.1255*

Table 19.5: Pruned results similarity statistics for tests using the *Melanoma* data set.

Test run	*mean(sim)*	*stddev(sim)*	*mean(maxSim)*	*stddev(maxSim)*
1	0.5971	0.3605	0.7404	0.3164
2	0.6417	0.3318	0.7188	0.3041
3	0.5047	0.2497	0.5769	0.2276
4	0.5662	0.2253	0.6796	0.2529
5	0.5442	0.3104	0.7662	0.1708
avg.	*0.5708*	*0.2955*	*0.6964*	*0.2544*

Table 19.6: Pruned results similarity statistics for tests using the *Wisconsin* data set.

Test run	*mean(sim)*	*stddev(sim)*	*mean(maxSim)*	*stddev(maxSim)*
1	0.5832	0.1332	0.6463	0.1015
2	0.5544	0.2168	0.5517	0.2099
3	0.4977	0.1473	0.6073	0.1331
4	0.6065	0.1018	0.7159	0.0857
5	0.5394	0.1987	0.6766	0.1213
avg.	*0.5562*	*0.1596*	*0.6396*	*0.1303*

Table 19.7: Pruned results similarity statistics for tests using the *Thyroid* data set.

Test run	*mean(sim)*	*stddev(sim)*	*mean(maxSim)*	*stddev(maxSim)*
1	0.5605	0.1488	0.6329	0.0918
2	0.5510	0.1415	0.6679	0.0555
3	0.5577	0.1448	0.6290	0.0931
4	0.5494	0.1159	0.6528	0.0286
5	0.4647	0.2023	0.6626	0.1077
avg.	*0.5367*	*0.1507*	*0.6490*	*0.0753*

Table 19.8: Pruned results similarity statistics for tests using the NO_x data set.

Test run	*mean(sim)*	*stddev(sim)*	*mean(maxSim)*	*stddev(maxSim)*
1	0.6049	0.1995	0.7774	0.1853
2	0.5731	0.1479	0.7089	0.0750
3	0.6055	0.1217	0.7252	0.0757
4	0.6435	0.1352	0.7389	0.0940
5	0.6921	0.0926	0.7702	0.0752
avg.	*0.6238*	*0.1394*	*0.7441*	*0.1010*

19.4 Conclusion

In this chapter we have discussed results regarding the comparison of models produced by evolutionary system identification based on genetic programming; using this similarity estimation we have systematically compared the results obtained for the identification of NO_x emissions of a motor engine as well as medical benchmark machine learning data sets. Comparing these results we see that especially the pruned models, i.e. those that have been deprived of genetic junk, are a lot more similar to each other. Thus, the results presented here can be seen as another indication that GP using enhanced selection concepts does not only work, but also has to be considered reliable with respect to comparability and similarity of results if there really is a system whose behavior is represented in the given training data.

Part III

Conclusion

Chapter 20

Conclusion and Future Perspectives

In this thesis several techniques have been described that are able to enhance the power of genetic programming (GP) in data based modeling. In addition to this, several concepts for monitoring what is going on in the population of a genetic programming process have been presented; time series data representing measurements of mechatronical systems have been used as well as classification data sets for demonstrating how these GP concepts effect the GP process.

In Part I we have summarized the basics of evolutionary computation, GP in general and concepts for selection and parallelization. Data based modeling has also been described as well as the GP implementation for the HeuristicLab framework; time series and classification analysis specific aspects for GP have been explored as well as concepts for local adaptation and the introduction of a priori knowledge into the GP process. Estimation models for the similarity of GP solution candidates have also been introduced; these similarity calculations are (amongst others) used for monitoring population dynamics in GP processes. Part I is concluded by a discussion of on-line and sliding window genetic programming.

Several test series have been summarized in Part II; these empirical test studies have been executed and analyzed in order to demonstrate how and how well evolutionary system identification works using the concepts described in the first part of this thesis. The most interesting results of these empirical test series can be summarized in the following way:

- The use of enhanced selection models has significantly positive effects on GP's ability to produce high quality models. For time series as well as classification data sets we have shown cases in which GP using a combination of proportional and random parents selection and strict offspring selection was able to generate models that perform better in terms of achievable solution quality and stability than those produced using standard GP.

- Sliding window GP, implemented as a generalization of an on-line GP simulation, can be used in order to combat known problems of GP based system identification such as overfitting and rather high runtime consumption.

- Genetic propagation has been investigated by analyzing which parts of the population are able to place children into the next generation's pool of individuals.

- Population diversity has been investigated in single population GP as well as in multi-population GP implementations. We have shown that population diversity in standard GP variants differs significantly from diversity progresses in standard GP implementations; this can be seen as a result of offspring selection and the migration frequencies applied in multi-population GP using various migration concepts.

- The use of pruning strategies as well as fitness based punishment of large models and the use of fixed complexity limits have been analyzed as strategies for combating code bloat. This can also be seen in the context of the avoidance of premature convergence: By removing parts of model structures that are redundant or do not contribute significantly to the models' performance, there is more space available that can be used for building better (i.e., more meaningful) subtree structures.

- Inserting (parts of) models into the GP process as well as extending the functions library and enabling the GP process to build models that use the functions that represent the available knowledge can also help the GP process to evolve reasonable models. Still, the introduction of a priori knowledge into the GP process can be dangerous as there can be a significant loss of genetic diversity if models representing available knowledge are introduced directly into the GP population with too high frequency.

- Finally, we have analyzed the similarity of models produced by GP in separate test runs. Using a structural similarity estimation function we have systematically compared the results obtained for the identification of NO_x emissions of a motor engine as well as medical benchmark machine learning data sets. Comparing these results we see that especially the pruned models, i.e. those that have been deprived of genetic junk, show a rather high similarity with each other. These results can be seen as another indication that GP using enhanced selection concepts does not only work, but also has to be considered reliable with respect to comparability and similarity of results if there really is a system whose behavior is represented in the given training data.

We hope that the use of the possibilities for monitoring populations, that have been presented in this thesis, can help to analyze how and why modifications of the standard GP process affect GP populations. For example, analyzing the genetic propagation and population diversity results for extended GP gives hints why the use of gender specific parents selection and offspring selection enables genetic programming to produce even better models than standard genetic programming. The implicit ability of GP to perform the selection of relevant variables in combination with structure identification and parameters optimization is also more obvious when using extended GP based techniques. In any case, we have presented graphical representations of the analysis results that are intuitive and easily understandable; we are convinced that the information provided by these GP population analysis features can help designers as well as users of GP implementations to tune their GP applications and thus produce even better results.

Of course, several concepts presented here can be used not only for structure identification, but also for other GP applications; in fact, several aspects are designed in such a generic way that they can be transferred to any GA application. This is why we suggest transferring these concepts to other application areas of evolutionary computation. Furthermore, even though we have discussed the use of several ways how to monitor internal processes in GP populations, we have not yet designed or implemented methods that use this information for steering GP processes; this surely should be done in the near future.

Part IV

Indices

Bibliography

[AA05] Wendy Ashlock and Dan Ashlock. Single parent genetic programming. In David Corne, Zbigniew Michalewicz, Marco Dorigo, Gusz Eiben, David Fogel, Carlos Fonseca, Garrison Greenwood, Tan Kay Chen, Guenther Raidl, Ali Zalzala, Simon Lucas, Ben Paechter, Jennifier Willies, Juan J. Merelo Guervos, Eugene Eberbach, Bob McKay, Alastair Channon, Ashutosh Tiwari, L. Gwenn Volkert, Dan Ashlock, and Marc Schoenauer, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 2, pages 1172–1179, Edinburgh, UK, 2-5 September 2005. IEEE Press.

[AdRWL05] Daniel Alberer, Luigi del Re, Stephan Winkler, and Peter Langthaler. Virtual sensor design of particulate and nitric oxide emissions in a DI diesel engine. In *Proceedings of the 7th International Conference on Engines for Automobile ICE 2005*, number 2005-24-063, 2005.

[Aff01] Michael Affenzeller. Transferring the concept of selective pressure from evolutionary strategies to genetic algorithms. In Z. Bubnicki and A. Grzech, editors, *Proceedings of the 14th International Conference on Systems Science*, volume 2, pages 346–353. Oficyna Wydawnicza Politechniki Wroclawskiej, 2001.

[Aff03] Michael Affenzeller. *New Hybrid Variants of Genetic Algorithms: Theoretical and Practical Aspects.* Schriften der Johannes Kepler Universität Linz. Universitätsverlag Rudolf Trauner, 2003.

[Aff05] Michael Affenzeller. *Population Genetics and Evolutionary Computation: Theoretical and Practical Aspects.* Schriften der Johannes Kepler Universität Linz. Universitätsverlag Rudolf Trauner, 2005.

[AK95] David Andre and John R. Koza. Parallel genetic programming on a network of transputers. In Justinian P. Rosca, editor, *Proceedings* *of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 111–120, Tahoe City, California, USA, 9 July 1995.

[AK96] David Andre and John R. Koza. Parallel genetic programming: A scalable implementation using the transputer network architecture. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 16, pages 317–338. MIT Press, Cambridge, MA, USA, 1996.

[Alb05] Enrique Alba. *Parallel Metaheuristics: A New Class of Algorithms.* Wiley-Interscience, 2005.

[Alt94a] Lee Altenberg. Emergent phenomena in genetic programming. In Anthony V. Sebald and Lawrence J. Fogel, editors, *Evolutionary Programming — Proceedings of the Third Annual Conference*, pages 233–241, San Diego, CA, USA, 24-26 February 1994. World Scientific Publishing.

[Alt94b] Lee Altenberg. The Schema Theorem and Price's Theorem. In L. Darrell Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 23–49, Estes Park, Colorado, USA, 31 July–2 August 1994. Morgan Kaufmann. Published 1995.

[And71] Theodore W. Anderson. *The Statistical Analysis of Time Series.* Wiley, 1971.

[And76] Oliver D. Anderson. *Time Series Analysis and Forecasting: the Box-Jenkins Approach.* Butterworth, 1976.

[Ang93] Peter John Angeline. *Evolutionary Algorithms and Emergent Intelligence.* PhD thesis, Ohio State University, 1993.

[Ang94] Peter John Angeline. Genetic programming and emergent intelligence. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 4, pages 75–98. MIT Press, 1994.

[Ang96] Peter J. Angeline. Genetic programming's continued evolution. In Kenneth E. Kinnear and Peter J. Angeline, editors, *Advances in Genetic Programming 2*, pages 1 – 20. MIT Press, Cambridge, MA, USA, 1996.

[Ang98] Peter J. Angeline. Subtree crossover causes bloat. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco

Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 745–752, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[AT99] Enrique Alba and José M. Troya. A survey of parallel distributed genetic algorithms. *Complexity (USA)*, 4(4):31–52, 1999.

[AW04] Michael Affenzeller and Stefan Wagner. SASEGASA: A new generic parallel evolutionary algorithm for achieving highest quality results. *Journal of Heuristics - Special Issue on New Advances on Parallel Meta-Heuristics for Complex Problems*, 10:239–263, 2004.

[AWW05a] Michael Affenzeller, Stefan Wagner, and Stephan Winkler. GA-selection revisited from an ES-driven point of view. In J. Mira and J. R. Alvarez, editors, *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach*, volume 3562 of *Lecture Notes in Computer Science*, pages 262–271. Springer, 2005.

[AWW05b] Michael Affenzeller, Stefan Wagner, and Stephan Winkler. Goal-oriented preservation of essential genetic information by offspring selection. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) 2005*, volume 2, pages 1595–1596. Association for Computing Machinery (ACM), 2005.

[BB94] Marco Bohanec and Ivan Bratko. Trading accuracy for simplicity in decision trees. *Machine Learning*, 15:223 – 250, 1994.

[BD91] Peter J. Brockwell and Richard A. Davis. *Time Series: Theory and Methods*. Springer, 1991.

[BD96] Peter J. Brockwell and Richard A. Davis. *A First Course in Time Series Analysis*. Springer, 1996.

[BES01] Elizabeth Bradley, Matthew Easley, and Reinhard Stolle. Reasoning about nonlinear system identification. *Artificial Intelligence*, 133:139–188, December 2001.

[BGK04] Edmund K. Burke, Steven Gustafson, and Graham Kendall. Diversity in genetic programming: An analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation*, 8(1):47–62, 2004.

[BJ76] George E. P. Box and Gwilym M. Jenkins. *Time Series Analysis: Forecasting and Control*. Holden-Day, 1976.

[BK00] Vladan Babovic and Maarten Keijzer. Genetic programming as a model induction engine. *Journal of Hydroinformatics*, 1(2):35–60, 2000.

[BKSS99] Forrest H Bennett III, John R. Koza, James Shipman, and Oscar Stiffelman. Building a parallel computer system for $18,000 that performs a half peta-flop per day. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1484–1490, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[BL04] Wolfgang Banzhaf and Christian W. G. Lasarczyk. Genetic programming of an algorithmic chemistry. In Una-May O'Reilly, Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 11, pages 175–190. Springer, Ann Arbor, 13-15 May 2004.

[BLFM04] Celia C. Bojarczuk, Heitor S. Lopes, Alex A. Freitas, and Edson L Michalkiewicz. A constrained-syntax genetic programming system for discovering classification rules: application to medical data sets. *Artificial Intelligence in Medicine*, 30(1):27–48, January 2004.

[BNKF98] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, January 1998.

[BOA$^+$05] Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors. *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, Washington DC, USA, 25-29 June 2005. ACM Press.

[Bra97] Andrew Bradley. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30:1145–1159, 1997.

[BU95] Carla E. Brodley and Paul E. Utgoff. Multivariate decision trees. *Machine Learning*, 19(1):45–77, 1995.

[CG93] Helen G. Cobb and John Grefenstette. Genetic algorithms for tracking changing environments. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 523–530, 1993.

[Cha01] Chris Chatfield, editor. *Time Series and Forecasting.* Chapman and Hall, 2001.

[CJ91a] Robert J. Collins and David R. Jefferson. Antfarm: Towards simulated evolution. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, pages 579–601. Addison-Wesley, Redwood City, CA, 1991.

[CJ91b] Robert J. Collins and David R. Jefferson. Representations for artificial organisms. In Jean-Arcady Meyer and Stewart W. Wilson, editors, *Proceedings of the First International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 382–390. MIT Press, 1991.

[CO07] Steffen Christensen and Franz Oppacher. Solving the artificial ant on the santa fe trail problem in 20,696 fitness evaluations. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1574–1579, London, 7-11 July 2007. ACM Press.

[CP94] Weon Sam Chung and Rafael A. Perez. The schema theorem considered insufficient. *Proceedings of the Sixth IEEE International Conference on Tools with Artificial Intelligence*, pages 748–751, 1994.

[CP97] Erick Cantú-Paz. A survey of parallel genetic algorithms. Technical Report IlliGAL 97003, University of Illinois at Urbana-Champaign, 1997.

[CP01] Erick Cantú-Paz. *Efficient and Accurate Parallel Genetic Algorithms.* Kluwer Academic Publishers, 2001.

[CPFD+03a] Erick Cantú-Paz, James A. Foster, Kalyanmoy Deb, Lawrence Davis, Rajkumar Roy, Una-May O'Reilly, Hans-Georg Beyer, Russell K. Standish, Graham Kendall, Stewart W. Wilson, Mark Harman, Joachim Wegener, Dipankar Dasgupta, Mitchell A. Potter, Alan C. Schultz, Kathryn A. Dowsland, Natasha Jonoska, and Julian F. Miller, editors. *Genetic and Evolutionary Computation – GECCO 2003, Part I*, volume 2723 of *Lecture Notes in Computer Science*, Chicago, IL, USA, 12-16 July 2003. Springer.

[CPFD+03b] Erick Cantú-Paz, James A. Foster, Kalyanmoy Deb, Lawrence Davis, Rajkumar Roy, Una-May O'Reilly, Hans-Georg Beyer, Russell K. Standish, Graham Kendall, Stewart W. Wilson, Mark Harman, Joachim Wegener, Dipankar Dasgupta, Mitchell A. Potter, Alan C. Schultz, Kathryn A. Dowsland, Natasha Jonoska, and Julian F. Miller, editors. *Genetic and Evolutionary Computation – GECCO 2003, Part II*, volume 2724 of *Lecture Notes in Computer Science*. Springer, 12-16 July 2003.

[CPG99] Erick Cantu-Paz and David E. Goldberg. On the scalability of parallel genetic algorithms. *Evolutionary Computation*, 7(4):429–449, 1999.

[Cra85] Nichael L. Cramer. A representation for the adapative generation of simple sequential programs. *International Conference on Genetic Algorithms and their Applications (ICGA85)*, pages 183–187, 1985.

[CTE+06] Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, editors. *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, Budapest, Hungary, 10 - 12 April 2006. Springer.

[DAG01] Wlodzislaw Duch, Rafal Adamczak, and Krzysztof Grabczewski. A new methodology of extraction, optimization and application of crisp and fuzzy logical rules. *IEEE Transactions on Neural Networks*, 12:277–306, 2001.

[Dar59] Charles Darwin. *The Origin of Species By Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life.* Murray, London, 1859.

[Dar98] Charles Darwin. *The Origin of Species.* Wordsworth Classics of World Literature. Wordsworth Editions Limited, 1998.

[Dei04] Manfred Deistler. System identification and time series analysis: Past, present, and future. In *Stochastic Theory and Control: Proceedings of a Workshop held in Lawrence, Kansas*, Lecture Notes in Control and Information Sciences, pages 97–110. Springer Berlin / Heidelberg, 2004.

[DG89] Kalyanmoy Deb and David E. Goldberg. An investigation of niche and species formation in genetic function optimization. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 42–50. Morgan Kaufmann, 1989.

[DH02] Judith E. Devaney and John G. Hagedorn. The role of genetic programming in describing the microscopic structure of hydrating plaster. In Erick Cantú-Paz, editor, *Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002)*, pages 91–98, New York, NY, July 2002. AAAI.

[DHS00] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley Interscience, 2^{nd} edition, 2000.

[DLJD00] Dumitru Dumitrescu, Beatrice Lazzerini, Lakhmi C. Jain, and Adrian Dumitrescu. *Evolutionary Computation*. The CRC Press International Series on Computational Intelligence. CRC Press, 2000.

[dN06] Lilian M. de Menezes and Nikolay Y. Nikolaev. Forecasting with genetically programmed polynomial neural networks. *International Journal of Forecasting*, 22(2):249–265, April-June 2006.

[DOMK$^+$01] Stephan Dreiseitl, Lucila Ohno-Machado, Harald Kittler, Staal Vinterbo, Holger Billhardt, and Michael Binder. A comparison of machine learning methods for the diagnosis of pigmented skin lesions. *Journal of Biomedical Informatics*, 34:28–36, 2001.

[DPB$^+$04a] Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund Burke, Paul Darwen, Dipankar Dasgupta, Dario Floreano, James Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andy Tyrrell, editors. *Genetic and Evolutionary Computation – GECCO-2004, Part I*, volume 3102 of *Lecture Notes in Computer Science*, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[DPB$^+$04b] Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund Burke, Paul Darwen, Dipankar Dasgupta, Dario Floreano, James Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andy Tyrrell, editors. *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[dRLF$^+$05] Luigi del Re, Peter Langthaler, Christian Furtmüller, Stephan Winkler, and Michael Affenzeller. NOx virtual sensor based on structure identification and global optimization. In *Proceedings of the SAE World Congress 2005*, number 2005-01-0050, 2005.

[Dro98] Stefan Droste. Genetic programming with guaranteed quality. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 54–59, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[DS98] Norman R. Draper and Harry Smith. *Applied Regression Analysis.* Wiley, 1998.

[Eic07] Christoph F. Eick. *Evolutionary Programming: Genetic Programming* (http://www2.cs.uh.edu/∼ceick/6367/eiben6.ppt, class transparencies). Department of Computer Science, University of Houston, Texas, 2007.

[EKK04] Jeroen Eggermont, Joost N. Kok, and Walter A. Kosters. Detecting and pruning introns for faster decision tree evolution. In Xin Yao, Edmund Burke, Jose A. Lozano, Jim Smith, Juan J. Merelo-Guervós, John A. Bullinaria, Jonathan Rowe, Peter Tiňo Ata Kabán, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *LNCS*, pages 1071–1080, Birmingham, UK, 18-22 September 2004. Springer-Verlag.

[Ell98] Jeff Ellis. An investigation of predictive and adaptive model-based methods for direct ground-to-space teleoperation with time delay. Master's thesis, Wright State University, 1998.

[EN00] Aniko Ekart and S. Z. Nemeth. A metric for genetic programs and fitness sharing. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 259–270, Edinburgh, 15-16 April 2000. Springer-Verlag.

[EN01] Aniko Ekart and S. Z. Nemeth. Selection based on the pareto non-domination criterion for controlling code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 2(1):61–73, March 2001.

[EOE$^+$07] Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors. *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, Valencia, Spain, 11 - 13 April 2007. Springer.

[ES03] Agoston E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[FBF$^+$03] Peter Flach, Hendrik Blockeel, Cesar Ferri, José Hernández-Orallo, and Jan Struyf. Decision support for data mining: Introduction to ROC analysis and its applications. *Data mining and decision support: Integration and collaboration*, 2003.

[FE05] Jonathan E. Fieldsend and Richard M. Everson. Formulation and comparison of multi-class ROC surfaces. *Proceedings of the ICML 2005 Workshop on ROC Analysis in Machine Learning*, pages 41–48, 2005.

[FG97] David B. Fogel and Adam Ghozeil. Schema processing under proportional selection in the presence of random effects. *IEEE Trans. Evolutionary Computation*, 1(4):290–293, 1997.

[FG98] David B. Fogel and Adam Ghozeil. The schema theorem and the misallocation of trials in the presence of stochastic effects. *Proceedings of the 7th International Conference on Evolutionary Programming VI*, 1447:313–321, 1998.

[Fog94] David B. Fogel. An introduction to simulated evolutionary optimization. *IEEE Trans. on Neural Networks*, 5(1):3–14, 1994.

[For81] Richard Forsyth. BEAGLE – A Darwinian approach to pattern recognition. *Kybernetes*, 10:159–166, 1981.

[Fox97] John Fox. *Applied Regression Analysis, Linear Models and Related Methods*. Sage, 1997.

[FP98] Pablo Funes and Jordan Pollack. Evolutionary body building: Adaptive physical designs for robots. *Artificial Life*, 4(4):337–357, Fall 1998.

[FPS06] Gianluigi Folino, Clara Pizzuti, and Giandomenico Spezzano. Improving cooperative GP ensemble with clustering and pruning for pattern classification. In Maarten Keijzer, Mike Cattolico, Dirk Arnold, Vladan Babovic, Christian Blum, Peter Bosman, Martin V. Butz, Carlos Coello Coello, Dipankar Dasgupta, Sevan G. Ficici, James Foster, Arturo Hernandez-Aguirre, Greg Hornby, Hod Lipson, Phil McMinn, Jason Moore, Guenther Raidl, Franz Rothlauf, Conor Ryan, and Dirk Thierens, editors, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 1, pages 791–798, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[FPSS96] Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery: An overview. *Advances in Knowledge Discovery and Data Mining*, 1996.

[GAMRRP07] Marc Garcia-Arnau, Daniel Manrique, Juan Rios, and Alfonso Rodriguez-Paton. Initialization method for grammar-guided genetic programming. *Knowledge-Based Systems*, 20(2):127–133, March 2007.

[GAT06] Alma Lilia Garcia-Almanza and Edward P. K. Tsang. Simplifying decision trees learned by genetic programming. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 7906–7912, Vancouver, 6-21 July 2006. IEEE Press.

[GB89] John J. Grefenstette and James E. Baker. How genetic algorithms work: A critical look at implicit parallelism. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, 1989.

[GL97] Fred Glover and Fred Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[Glo86] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13:533–549, 1986.

[GMW82] Philip Gill, Walter Murray, and Margaret Wright. *Practical Optimization.* Academic Press, 1982.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley Longman, 1989.

[GR94] Chris Gathercole and Peter Ross. Dynamic training subset selection for supervised learning in genetic programming. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature III*, volume 866 of *LNCS*, pages 312–321, Jerusalem, 9-14 October 1994. Springer-Verlag.

[Gru94] Frederic Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Laboratoire de l'Informatique du Parallilisme, Ecole Normale Supirieure de Lyon, France, 1994.

[Gru96] Frederic Gruau. On using syntactic constraints with genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 19, pages 377–394. MIT Press, Cambridge, MA, USA, 1996.

[Ham58] Charles L. Hamblin. Computer languages. *The Australian Journal of Science*, 20:135–139, 1958.

[Ham62] Charles L. Hamblin. Translation to and from Polish notation. *Computer Journal*, 5:210–213, 1962.

[Ham94] James D. Hamilton. *Time Series Analysis.* Princeton University Press, 1994.

[Hey88] John B. Heywood. *Internal Combustion Engine Fundamentals.* McGraw Hill, 1988.

[HHM04] Hoang Tuan Hao, Nguyen Xuan Hoai, and Robert I McKay. Does it matter where you start? A comparison of two initialisation strategies for grammar guided genetic programming. In R I Mckay and Sung-Bae Cho, editors, *Proceedings of The Second Asian-Pacific Workshop on Genetic Programming*, Cairns, Australia, 6-7 December 2004.

[HLdVL07] José Ignacio Hidalgo, Juan Lanchares, Francisco Fernández de Vega, and Daniel Lombraña. Is the island model fault tolerant? In *Proceedings of the Genetic and Evolutionary Computation Conference*

GECCO 2007, pages 2737–2744. Association for Computing Machinery (ACM), 2007.

[HM82] James A. Hanley and Barbara J. McNeil. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology*, 143:29–36, 1982.

[HOFLF04] José Hernández-Orallo, César Ferri, Nicolas Lachiche, and Peter A. Flach, editors. *ROC Analysis in Artificial Intelligence, 1st International Workshop, ROCAI-2004, Valencia, Spain, August 22, 2004*, 2004.

[Hol75] John H. Holland. *Adaption in Natural and Artifical Systems.* University of Michigan Press, 1975.

[Här90] Wolfgang Härdle. *Applied Nonparametric Regression.* Cambridge University Press, 1990.

[HRv07] Kenneth Holladay, Kay Robbins, and Jeffery von Ronne. FIFTH: A stack based GP language for vector processing. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 102–113, Valencia, Spain, 11 - 13 April 2007. Springer.

[HS95] David P. Helmbold and Robert E. Schapire. Predicting nearly as well as the best pruning of a decision tree. *Proceedings of the Eighth Annual Conference on Computational Learning Theory*, pages 61 – 68, 1995.

[HSC96] Howard J. Hamilton, Ning Shan, and Nick Cercone. Riac: A rule induction algorithm based on approximate classification. Technical Report CS 96-06, Regina University, 1996.

[HSD01] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 97–106, 2001.

[HW07] Markus Hirsch and Stephan Winkler. Personal communication on combustion and formation of NO_x, November 2007.

[IIS98] Takuya Ito, Hitoshi Iba, and Satoshi Sato. Non-destructive depth-dependent crossover for genetic programming. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 71–82, Paris, 14-15 April 1998. Springer-Verlag.

[Jac94] Dale Jacquette. *Philosophy of Mind*. Prentice Hall, 1994.

[Jac99] Christian Jacob. Lindenmayer systems and growth program evolution. In Talib S. Hussain, editor, *Advanced Grammar Techniques Within Genetic Programming and Evolutionary Computation*, pages 76–79, Orlando, Florida, USA, 13 July 1999.

[JCC$^+$92] David Jefferson, Robert Collins, Claus Cooper, Michael Dyer, Margot Flowers, Richard Korf, Charles Taylor, and Alan Wang. Evolution as a theme in artificial life: The genesys/tracker system. *Artificial Life II*, pages 417–434, 1992.

[JHC04] Istvan Jonyer, Lawrence B. Holder, and Diane J. Cook. Attribute-value selection based on minimum description length. In *Proceedings of the International Conference on Artificial Intelligence*, pages 1154–1159, 2004.

[JM05] Xianhua Jiang and Yuichi Motai. Incremental on-line PCA for automatic motion learning of eigen behavior. *Proceedings of the 1st International Workshop on Automatic Learning and Real-Time ALaRT '05*, pages 153–164, 2005.

[Jon75] Kenneth A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975.

[JP96] Hugues Juille and Jordan B. Pollack. Massively parallel genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 17, pages 339–358. MIT Press, Cambridge, MA, USA, 1996.

[KB99] Maarten Keijzer and Vladan Babovic. Dimensionally aware genetic programming. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1069–1076, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[KBAK99] John Koza, Forrest H Bennett III, David Andre, and Martin A. Keane. The design of analog circuits by means of genetic programming. In Peter Bentley, editor, *Evolutionary Design by Computers*, chapter 16, pages 365–385. Morgan Kaufmann, San Francisco, USA, 1999.

[KCA$^+$06] Maarten Keijzer, Mike Cattolico, Dirk Arnold, Vladan Babovic, Christian Blum, Peter Bosman, Martin V. Butz, Carlos Coello Coello, Dipankar Dasgupta, Sevan G. Ficici, James Foster, Arturo Hernandez-Aguirre, Greg Hornby, Hod Lipson, Phil McMinn, Jason Moore, Guenther Raidl, Franz Rothlauf, Conor Ryan, and Dirk Thierens, editors. *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[Kei96] Maarten Keijzer. Efficiently representing populations in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 13, pages 259–278. MIT Press, Cambridge, MA, USA, 1996.

[Kei02] Maarten Keijzer. *Scientific Discovery using Genetic Programming.* PhD thesis, Danish Technical University, Lyngby, Denmark, March 2002.

[Ken73] Maurice G. Kendall. *Time Series.* Griffin, 1973.

[KGV83] Scott Kirkpatrick, Charles Daniel Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[KIAK99] John R. Koza, Forrest H. Bennett III, David Andre, and Martin A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving.* Morgan Kaufmann Publishers, 1999.

[Kin93] Kenneth E. Kinnear, Jr. Generality and difficulty in genetic programming: Evolving a sort. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 287–294, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[KKS$^+$03a] John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Learning.* Kluwer Academic Publishers, 2003.

[KKS$^+$03b] John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.* Kluwer Academic Publishers, 2003.

[Knu64] Donald E. Knuth. Backus normal form versus backus naur form. *Communications of the ACM*, 7:735–736, 1964.

[KO90] Maurice G. Kendall and J. Keith Ord. *Time Series.* Edward Arnold, 1990.

[KOL$^+$04] Maarten Keijzer, Una-May O'Reilly, Simon M. Lucas, Ernesto Costa, and Terence Soule, editors. *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, Coimbra, Portugal, 5-7 April 2004. Springer-Verlag.

[Koz89] John R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774. Morgan Kaufmann, 20-25 August 1989.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, 1992.

[Koz94] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* The MIT Press, 1994.

[KP98] Ron Kohavi and Foster Provost. Glossary of terms. *Machine Learning, Special Issue on Applications of Machine Learning and the Knowledge Discovery Process*, 30:271–274, 1998.

[KSBK01] Sathiya Keerthi, Shirish K. Shevade, Chiranjib Bhattacharyya, and Radha K. M. Karuturi. Improvements to platt's SMO algorithm for SVM classifier design. *Neural Computation*, 13(3):637–649, 2001.

[KTC$^+$05] Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors. *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.

[Kus98] Ibrahim Kuscu. Evolving a generalised behavior: Artificial ant problem revisited. In V. William Porto, N. Saravanan, D. Waagen, and

A. E. Eiben, editors, *Seventh Annual Conference on Evolutionary Programming*, volume 1447 of *LNCS*, pages 799–808, Mission Valley Marriott, San Diego, California, USA, 25-27 March 1998. Springer-Verlag.

[Lan95] William B. Langdon. Evolving data structures using genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 295–302, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Lan98] William B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming*. Kluwer, Boston, 24 April 1998.

[Lan99] William B. Langdon. Size fair tree crossovers. In Eric Postma and Marc Gyssen, editors, *Proceedings of the Eleventh Belgium/Netherlands Conference on Artificial Intelligence (BNAIC'99)*, pages 255–256, Kasteel Vaeshartelt, Maastricht, Holland, 3-4 November 1999.

[Lan00] William B. Langdon. Size fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines*, 1(1/2):95–119, April 2000.

[LAWdR05] Peter Langthaler, Daniel Alberer, Stephan Winkler, and Luigi del Re. Design eines virtuellen Sensors für Partikelmessung am Dieselmotor. In M. Horn, M. Hofbauer, and N. Dourdoumas, editors, *Proceedings of the 14th Styrian Seminar on Control Engineering and Process Automation (14. Steirisches Seminar über Regelungstechnik und Prozessautomatisierung)*, pages 71–87, 2005.

[LC01] Thomas Loveard and Victor Ciesielski. Representing classification problems in genetic programming. In *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1070–1077, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 2001. IEEE Press.

[Lev44] Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. *The Quarterly of Applied Mathematics*, 2:164–168, 1944.

[Lev66] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[LH06] Peter Lichodzijewski and Malcolm I. Heywood. Pareto-coevolutionary genetic programming for problem decomposition in multi-class classification. *Proceedings of the Genetic and Evolutionary Computation Conference GECCO'07*, pages 464–471, 2006.

[Lju99] Lennart Ljung. *System Identification – Theory For the User, 2nd edition.* PTR Prentice Hall, Upper Saddle River, N.J., 1999.

[LN00] William B. Langdon and Peter Nordin. Seeding GP populations. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 304–315, Edinburgh, 15-16 April 2000. Springer-Verlag.

[LP97] William B. Langdon and Riccardo Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London, 23-27 June 1997.

[LP98] William B. Langdon and Riccardo Poli. Why ants are hard. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 193–201, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[LP02] William B. Langdon and Riccardo Poli. *Foundations of Genetic Programming.* Springer Verlag, Berlin Heidelberg New York, 2002.

[LS97] Sean Luke and Lee Spector. A comparison of crossover and mutation in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 240–248, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[LW95] Jack Y. B. Lee and P. C. Wong. The effect of function noise on GP efficiency. In Xin Yao, editor, *Progress in Evolutionary Computation*, volume 956 of *Lecture Notes in Artificial Intelligence*, pages 1–16. Springer-Verlag, Heidelberg, Germany, 1995.

[Man97] Yishay Mansour. Pessimistic decision tree pruning based on tree size. *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 195–201, 1997.

[Mar63] Donald W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *SIAM J. Appl. Math.*, 11:431–441, 1963.

[McC60] John L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960.

[McK00] R I (Bob) McKay. Fitness sharing in genetic programming. In Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 435–442, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[MH99] Nicholas Freitag McPhee and Nicholas J. Hopper. Analysis of genetic diversity through population history. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1112–1120, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[MIB$^+$00] Katharina Morik, Michael Imboff, Peter Brockhausen, Thorsten Joachims, and Ursula Gather. Knowledge discovery and knowledge validation in intensive care. *Artificial Intelligence in Medicine*, 19(3):225–249, 2000.

[Mic92] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 1992.

[Min89] John Mingers. An empirical comparison of pruning methods for decision tree induction. *Machine Learning*, 4:227 – 243, 1989.

[Mit00] Tom Mitchell. *Machine Learning*. McGraw-Hill, New York, 2000.

[MJK07] Douglas C. Montgomery, Cheryl L. Jennings, and Murat Kulahci. *Introduction to Time Series Analysis and Forecasting*. Wiley & Sons, 2007.

[MK00] Yoichiro Maeda and Satomi Kawaguchi. Redundant node pruning and adaptive search method for genetic programming. In Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, page 535, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[Mor91] Foster Morrison. *The Art of Modeling Dynamic Systems: Forecasting for Chaos, Randomness, and Determinism.* John Wiley & Sons, Inc, 1991.

[MP43] Warren McCulloch and Walter Pitts. A logical calculus of the ideas imminent in nervous activity. In *Bulletin of Mathematical Biophysics*, volume 5, pages 115–137, 1943.

[NB95] Peter Nordin and Wolfgang Banzhaf. Complexity compression and evolution. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Nel01] Oliver Nelles. *Nonlinear System Identification.* Springer Verlag, Berlin Heidelberg New York, 2001.

[Nol97] Daniel Nolan. Quantitative parsimony. *British Journal for the Philosophy of Science*, 48(3):329–343, 1997.

[Nor97] Peter Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications.* PhD thesis, der Universitat Dortmund am Fachereich Informatik, 1997.

[Nør00] Magnus Nørgaard. Neural network based system identification toolbox. Technical Report 00-E-891, Technical University of Denmark, 2000.

[NV92] Allen E. Nix and Michael D. Vose. Modeling genetic algorithms with markov chains. *Annals of Mathematics and Artificial Intelligence*, 5(1):79–88, 1992.

[OO94] Una-May O'Reilly and Franz Oppacher. The troubling aspects of a building block hypothesis for genetic programming. In L. Darrell Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 73–88, Estes Park, Colorado, USA, 31 July–2 August 1994. Morgan Kaufmann. Published 1995.

[O'R95] Una-May O'Reilly. *An Analysis of Genetic Programming.* PhD thesis, Carleton University, Ottawa-Carleton Institute for Computer Science, Ottawa, Ontario, Canada, 22 September 1995.

[O'R97] Una-May O'Reilly. Using a distance metric on genetic programs to understand genetic operators. In *IEEE International Conference on Systems, Man, and Cybernetics, Computational Cybernetics and Simulation*, volume 5, pages 4092–4097, Orlando, Florida, USA, 12-15 October 1997.

[Pan83] Alan Pankratz. *Forecasting With Univariate Box-Jenkins Models: Concepts and Cases.* Wiley, 1983.

[Pan91] Alan Pankratz. *Forecasting With Dynamic Regression Models.* Wiley, 1991.

[Per94] Tim Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[PL97a] Riccardo Poli and William B. Langdon. An experimental analysis of schema creation, propagation and disruption in genetic programming. In Thomas Back, editor, *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 18–25, Michigan State University, East Lansing, MI, USA, 19-23 July 1997. Morgan Kaufmann.

[PL97b] Riccardo Poli and William B. Langdon. Genetic programming with one-point crossover. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 180–189. Springer-Verlag London, 23-27 June 1997.

[PL97c] Riccardo Poli and William B. Langdon. A new schema theory for genetic programming with one-point crossover and point mutation. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 278–285, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Pla99] John Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schoelkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*, pages 185–208. MIT Press, 1999.

[PLC05] Daniel Parrot, Xiaodong Li, and Vic Ciesielski. Multi-objective techniques in genetic programming for evolving classifiers. *Proceedings of the 2005 Congress on Evolutionary Computation (CEC '05)*, pages 183–190, 2005.

[PM01a] Riccardo Poli and Nicholas F. McPhee. Exact GP schema theory for headless chicken crossover and subtree mutation. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 1062–1069, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.

[PM01b] Riccardo Poli and Nicholas Freitag McPhee. Exact schema theorems for GP with one-point and standard crossover operating on linear structures and their application to the study of the evolution of size. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 126–142, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[PM01c] Riccardo Poli and Nicholas Freitag McPhee. Exact schema theory for GP and variable-length GAs with homologous crossover. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 104–111, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[PM03a] Riccardo Poli and Nicholas Freitag McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part I. *Evolutionary Computation*, 11(1):53–66, March 2003.

[PM03b] Riccardo Poli and Nicholas Freitag McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part II. *Evolutionary Computation*, 11(2):169–206, June 2003.

[PMR04] Riccardo Poli, Nicholas Freitag McPhee, and Jonathan E. Rowe. Exact schema theory and markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover. *Genetic Programming and Evolvable Machines*, 5(1):31–70, March 2004.

[Pol97] Riccardo Poli. Evolution of graph-like programs with parallel distributed genetic programming. In Thomas Back, editor, *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 346–353, Michigan State University, East Lansing, MI, USA, 19-23 July 1997. Morgan Kaufmann.

[Pol99a] Riccardo Poli. New results in the schema theory for GP with one-point crossover which account for schema creation, survival and disruption. Technical Report CSRP-99-18, University of Birmingham, School of Computer Science, December 1999.

[Pol99b] Riccardo Poli. Parallel distributed genetic programming. In David Corne, Marco Dorigo, and Fred Glover, editors, *New Ideas in Optimization*, Advanced Topics in Computer Science, chapter 27, pages 403–431. McGraw-Hill, Maidenhead, Berkshire, England, 1999.

[Pol00a] Riccardo Poli. Exact schema theorem and effective fitness for GP with one-point crossover. In Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 469–476, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[Pol00b] Riccardo Poli. Hyperschema theory for GP with one-point crossover, building blocks, and some new results in GA theory. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 163–180, Edinburgh, 15-16 April 2000. Springer-Verlag.

[Pol00c] Riccardo Poli. A macroscopic exact schema theorem and a redefinition of effective fitness for GP with one-point crossover. Technical Report CSRP-00-1, University of Birmingham, School of Computer Science, February 2000.

[Pol01] Riccardo Poli. Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover. *Genetic Programming and Evolvable Machines*, 2(2):123–163, June 2001.

[Pop92] Karl Popper. *The Logic of Scientific Discovery.* Taylor & Francis, 1992.

[PP01] Fabio Previdi and Thomas Parisini. Model-free fault detection: a spectral estimation approach based on coherency functions. *International Journal of Control*, 74:1107–1117, 2001.

[PTT01] Daniel Peña, George C. Tiao, and Ruey S. Tsay. *A Course in Time Series Analysis.* Wiley, 2001.

[Que03] Christian Queinnec. *LISP in Small Pieces.* Cambridge University Press, 2003.

[RB96] Justinian P. Rosca and Dana H. Ballard. Discovery of subroutines in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–202. MIT Press, Cambridge, MA, USA, 1996.

[Rec73] Ingo Rechenberg. *Evolutionsstrategie.* Friedrich Frommann Verlag, 1973.

[RF99] José Luis Rodríguez-Fernández. Ockham's razor. *Endeavour*, 23:121–125, 1999.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2^{nd} edition, 2003.

[Ros95a] Justinian Rosca. Towards automatic discovery of building blocks in genetic programming. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 78–85, MIT, Cambridge, MA, USA, 10–12 November 1995. AAAI.

[Ros95b] Justinian P. Rosca. Entropy-driven adaptive representation. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 23–32, Tahoe City, California, USA, 9 July 1995.

[Ros97] Justinian P. Rosca. Analysis of complexity drift in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 286–294, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[RS01] Alain Ratle and Michele Sebag. Avoiding the bloat with probabilistic grammar-guided genetic programming. In P. Collet, C. Fonlupt, J.-K. Hao, E. Lutton, and M. Schoenauer, editors, *Artificial Evolution 5th International Conference, Evolution Artificielle, EA 2001*, volume 2310 of *LNCS*, pages 255–266, Creusot, France, October 29-31 2001. Springer Verlag.

[RS03] Alain Ratle and Michele Sebag. A novel approach to machine discovery: Genetic programming and stochastic grammars. In S. Matwin and C. Sammut, editors, *Proceedings of Twelfth International Conference on Inductive Logic Programming*, volume 2583 of *LNCS*, pages 207–222, Sydney, Australia, July 9-11, 2002 2003. Springer Verlag.

[Sam59] Arthur Samuel. Some studies in machine learning using the game of checkers. In *IBM Journal of Research and Development*, volume 3, pages 211 – 229, 1959.

[Sch75] Hans-Paul Schwefel. *Evolutionsstrategie und numerische Optimierung*. PhD thesis, Technische Universität Berlin, 1975.

[Sch94] Hans-Paul Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*. Birkhäuser Verlag, Basel, Switzerland, 1994.

[Sei86] John H. Seinfeld. *Atmospheric Chemistry and Physics of Air Pollution*. Wiley, New York, 1986.

[SF98] Terence Soule and James A. Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *1998 IEEE International Conference on Evolutionary Computation*, pages 781–186, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.

[SFD96] Terence Soule, James A. Foster, and John Dickinson. Code growth in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[SG99] Kim Sterelny and Peter E. Griffiths. *Sex and Death: An Introduction to Philosophy of Biology*. University of Chicago Press, 1999.

[SH98] Peter W. H. Smith and Kim Harries. Code growth, explicitly defined introns, and alternative selection schemes. *Evolutionary Computation*, 6(4):339–360, Winter 1998.

[SJB$^+$93] William M. Spears, Kenneth A. De Jong, Thomas Bäck, David B. Fogel, and Hugo de Garis. An overview of evolutionary computation. In P. Bradzil, editor, *Proceedings of the 1993 European Conference on Machine Learning*, Vienna, Austria, 1993. Springer-Verlag, Berlin, Heidelberg, New York.

[SJW92] Wolfram Schiffmann, Merten Joost, and Randolf Werner. Optimization of the backpropagation algorithm for training multilayer perceptrons. Technical Report 15, University of Koblenz, Institute of Physics, 1992.

[SJW93] Wolfram Schiffmann, Merten Joost, and Randolf Werner. Comparison of optimized backpropagation algorithms. *Proceedings of the European Symposium on Artificial Neural Networks ESANN '93*, pages 97–104, 1993.

[Smi80] Stephen F. Smith. *A Learning System Based on Genetic Adaptive Algorithms*. PhD thesis, University of Pittsburgh, 1980.

[SOG04] Kumara Sastry, Una-May O'Reilly, and David E. Goldberg. Population sizing for genetic programming based on decision making. In Una-May O'Reilly, Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 4, pages 49–65. Springer, Ann Arbor, 13-15 May 2004.

[SPWR02] Christopher R. Stephens, Riccardo Poli, Alden H. Wright, and Jonathan E. Rowe. Exact results from A coarse grained formulation of the dynamics of variable-length genetic algorithms. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 578–585, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[Sri99] Ashwin Srinivasan. Note on the location of optimal classifiers in n-dimensional ROC space. Technical Report PRG-TR-2-99, Oxford University Computing Laboratory, 1999.

[SS01] Marc Schoenauer and Michele Sebag. Using domain knowledge in evolutionary system identification. In K. C. Giannakoglou, D. Tsahalis, J. Periaux, K. Papailiou, and T. C. Fogarty, editors, *Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems*, Athens, 19-21 September 2001.

[SW97] Christopher R. Stephens and Henri Waelbroeck. Effective degrees of freedom in genetic algorithms and the block hypothesis. *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, pages 34–40, 1997.

[SW99] Christopher R. Stephens and Henri Waelbroeck. Schemata evolution and building blocks. *Evolutionary Computation*, 7(2):109–124, 1999.

[SWM91] Timothy Starkweather, Darrell Whitley, and Keith E. Mathias. Optimization using distributed genetic algorithms. *Parallel Problem Solving from Nature*, pages 176–185, 1991.

[TBB$^+$07] Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors. *GECCO 2007: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, London, UK, 7-11 July 2007. ACM Press.

[TG97] Ismail Taha and Joydeep Ghosh. Evaluation and ordering of rules extracted from feedforward networks. *Proceedings of the IEEE International Conference on Neural Networks*, pages 221–226, 1997.

[TH02] David Terrio and Malcolm I. Heywood. Directing crossover for reduction of bloat in GP. In W. Kinsner, A. Seback, and K. Ferens, editors, *IEEE CCECE 2003: IEEE Canadian Conference on Electrical and Computer Engineering*, pages 1111–1115. IEEE Press, 12-15 May 2002.

[THL94] James Ting-Ho-Lo. Synthetic approach to optimal filtering. *IEEE Transactions on Neural Networks*, 5:803–811, 1994.

[Tho18] William M. Thorburn. The myth of occam's razor. *Mind*, 27:345–353, 1918.

[Vap98] Vladimir N. Vapnik. *Statistical Learning Theory.* Wiley, New York, 1998.

[vBS04] Richard van Basshuysen and Fred Schäfer. *Internal Combustion Engine Handbook.* SAE International, 2004.

[VL91] Michael D. Vose and Gunar E. Liepins. Punctuated equilibria in genetic search. *Complex Systems*, 5:31–44, 1991.

[Vos99] Michael D. Vose. *The simple genetic algorithm: foundations and theory.* MIT Press, Cambridge, MA, 1999.

[WA04a] Stefan Wagner and Michael Affenzeller. HeuristicLab grid - a flexible and extensible environment for parallel heuristic optimization. In Z. Bubnicki and A. Grzech, editors, *Proceedings of the 15th International Conference on Systems Science*, volume 1, pages 289–296. Oficyna Wydawnicza Politechniki Wroclawskiej, 2004.

[WA04b] Stefan Wagner and Michael Affenzeller. HeuristicLab grid - a flexible and extensible environment for parallel heuristic optimization. *Journal of Systems Science*, 30(4):103–110, 2004.

[WA04c] Stefan Wagner and Michael Affenzeller. The HeuristicLab optimization environment. Technical report, Institute of Formal Models and Verification, Johannes Kepler University, Linz, Austria, 2004.

[WA05a] Stefan Wagner and Michael Affenzeller. HeuristicLab: A generic and extensible optimization environment. In B. Ribeiro, R. F. Albrecht, A. Dobnikar, D. W. Pearson, and N. C. Steele, editors, *Adaptive and Natural Computing Algorithms*, Springer Computer Science, pages 538–541. Springer, 2005.

[WA05b] Stefan Wagner and Michael Affenzeller. SexualGA: Gender-specific selection for genetic algorithms. In N. Callaos, W. Lesso, and E. Hansen, editors, *Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI) 2005*, volume 4, pages 76–81. International Institute of Informatics and Systemics, 2005.

[WAW04a] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Identifying nonlinear model structures using genetic programming techniques. In R. Trappl, editor, *Cybernetics and Systems 2004*, volume 1, pages 689–694. Austrian Society for Cybernetic Studies, 2004.

[WAW04b] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. New methods for the identification of nonlinear model structures based upon genetic programming techniques. In Z. Bubnicki and A. Grzech, editors, *Proceedings of the 15th International Conference on Systems Science*, volume 1, pages 386–393. Oficyna Wydawnicza Politechniki Wroclawskiej, 2004.

[WAW05a] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Genetic programming based model structure identification using on-line system data. In F. Barros, A. Bruzzone, C. Frydman, and N. Gambiasi, editors, *Proceedings of Conceptual Modeling and Simulation Conference CMS 2005*, pages 177–186. Frydman, LSIS, Université Paul Cézanne Aix Marseille III, 2005.

[WAW05b] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. New methods for the identification of nonlinear model structures based upon genetic programming techniques. *Journal of Systems Science*, 31(1):5–13, 2005.

[WAW06a] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Advances in applying genetic programming to machine learning, focussing on classification problems. In *Proceedings of the 9th International Workshop on Nature Inspired Distributed Computing NIDISC '06, part of the Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium IPDPS 2006*. IEEE, 2006.

[WAW06b] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Automatic data based patient classification using genetic programming. In R. Trappl, R. Brachman, R.A. Brooks, H. Kitano, D. Lenat, O. Stock, W. Wahlster, and M. Wooldridge, editors, *Cybernetics and Systems 2006*, volume 1, pages 251–256. Austrian Society for Cybernetic Studies, 2006.

[WAW06c] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. HeuristicModeler: A multi-purpose evolutionary machine learning algorithm and its applications in medical data analysis. In A. Bruzzone, A. Guasch, M. Piera, and J. Rozenblit, editors, *Proceedings of the International Mediterranean Modelling Multiconference I3M 2006*, pages 629–634. Piera, LogiSim, Barcelona, Spain, 2006.

[WAW06d] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Sets of receiver operating characteristic curves and their use in the evaluation of multi-class classification. In *Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2006*, volume 2, pages 1601–1602. Association for Computing Machinery (ACM), 2006.

[WAW06e] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Using enhanced genetic programming techniques for evolving classifiers in the context of medical diagnosis - an empirical study. In *Proceedings of the GECCO 2006 Workshop on Medical Applications of Genetic and Evolutionary Computation (MedGEC 2006).* Association for Computing Machinery (ACM), 2006.

[WAW07a] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Advanced genetic programming based machine learning. *Journal of Mathematical Modelling and Algorithms*, 6(3):455–480, 2007.

[WAW07b] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Selection pressure driven sliding window genetic programming. *Lecture Notes in Computer Science 4739: Computer Aided Systems Theory - EuroCAST 2007*, pages 789–795, 2007.

[WAW07c] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Variables diversity in systems identification based on extended genetic programming. *Proceedings of the* 16^th *International Conference on Systems Science*, 2:470–479, 2007.

[WAW08a] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Offspring selection and its effects on genetic propagation in genetic programming based system identification. In Robert Trappl, editor, *Cybernetics and Systems 2008*, volume 2, pages 549–554. Austrian Society for Cybernetic Studies, 2008.

[WAW08b] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. On the reliability of nonlinear modeling using enhanced genetic programming techniques. In *Proceedings of the Chaotic Modeling and Simulation Conference (CHAOS2008)*, 2008.

[WC99] Peter A. Whigham and Peter F. Crapper. Time series modelling using genetic programming: An application to rainfall-runoff models. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 5, pages 89–104. MIT Press, Cambridge, MA, USA, June 1999.

[WEA$^+$05] Stephan Winkler, Hajrudin Efendic, Michael Affenzeller, Luigi del Re, and Stefan Wagner. On-line modeling based on genetic programming. In *Proceedings of the 1st International Workshop on Automatic Learning and Real-Time (ALaRT'05)*, pages 119–130. Institute of Real-Time Learning Systems, University Siegen, Germany, 2005.

[WEA$^+$06] Stephan Winkler, Hajrudin Efendic, Michael Affenzeller, Luigi del Re, and Stefan Wagner. On-line modeling based on genetic programming. *International Journal on Intelligent Systems Technologies and Applications*, 2(2/3):255–270, 2006.

[WEdR06] Stephan Winkler, Hajrudin Efendic, and Luigi del Re. Quality pre-assessment in steel industry using data based estimators. In S. Cierpisz, K. Miskiewicz, and A. Heyduk, editors, *Proceedings of the IFAC Workshop MMM'2006 on Automation in Mining, Mineral and Metal Industry*. International Federation for Automatic Control, 2006.

[Wei06] William W. S. Wei. *Time Series Analysis – Univariate and Multivariate Methods*. Addison-Wesley, 2006.

[WF05] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, 2005.

[WH87] Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP*. Addison Wesley, 1987.

[Whi95] Peter A. Whigham. A schema theorem for context-free grammars. In *1995 IEEE Conference on Evolutionary Computation*, volume 1, pages 178–181, Perth, Australia, 29 November - 1 December 1995. IEEE Press.

[Whi96a] Peter A. Whigham. *Grammatical Bias for Evolutionary Learning*. PhD thesis, School of Computer Science, University College, University of New South Wales, Australian Defence Force Academy, Canberra, Australia, 14 October 1996.

[Whi96b] Peter A. Whigham. Search bias, language bias, and genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 230–237, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[WHMS03] Ju Wen-Hua, David Madigan, and Steven L. Scott. On bayesian learning of sparse classifiers. Technical Report 2003-08, Avaya Labs Research, 2003.

[Wig60] Eugene Wigner. The unreasonable effectiveness of mathematics in the natural sciences. In *Communications on Pure and Applied Mathematics*, volume XIII, pages 1–14. John Wiley & Sons, Inc, New York, 1960.

[Win04] Stephan Winkler. Identifying nonlinear model structures using genetic programming. Master's thesis, Johannes Kepler University, Linz, Austria, 2004.

[WK90] Sholom M. Weiss and Ioannis Kapouleas. An empirical comparison of pattern recognition, neural nets, and machine learning classification methods. In J. W. Shavlik and T. G. Dietterich, editors, *Readings in Machine Learning*, pages 177–183. Kaufmann, San Mateo, CA, 1990.

[WK96] Gerhard Widmer and Miroslav Kubat. Learning in the presence of concept drift and hidden contexts. *Machine Learning*, 23(2):69–101, 1996.

[WL96] Annie S. Wu and Robert K. Lindsay. A survey of intron research in genetics. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel, editors, *Parallel Problem Solving From Nature IV. Proceedings of the International Conference on Evolutionary Computation*, volume 1141 of *LNCS*, pages 101–110, Berlin, Germany, 22-26 September 1996. Springer-Verlag.

[WMD96] Jürgen Warnatz, Ulrich Maas, and Robert W. Dibble. *Combustion - Physical and Chemical Fundamentals, Modeling and Simulation, Experiments, Pollutant Formation.* Springer-Verlag, Heidelberg, 1996.

[WWP$^+$07] Stefan Wagner, Stephan Winkler, Erik Pitzer, Gabriel Kronberger, Andreas Beham, Roland Braune, and Michael Affenzeller. Benefits of plugin-based heuristic optimization software systems. *Lecture Notes in Computer Science 4739: Computer Aided Systems Theory - EuroCAST 2007*, pages 747–754, 2007.

[ZC93] Mark H. Zweig and Gregory Campbell. Receiver-operating characteristic (ROC) plots: A fundamental evaluation tool in clinical medicine. *Clinical Chemistry*, 39:561–577, 1993.

[Zha97] Byoung-Tak Zhang. A taxonomy of control schemes for genetic code growth. Position paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97, 20 July 1997.

[Zha00] Byoung-Tak Zhang. Bayesian methods for efficient genetic programming. *Genetic Programming and Evolvable Machines*, 1(3):217–242, July 2000.

[ZM96] Byoung-Tak Zhang and Heinz Mühlenbein. Adaptive fitness functions for dynamic growing/pruning of program trees. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 12, pages 241–256. MIT Press, Cambridge, MA, USA, 1996.

List of Tables

List of Figures

List of Algorithms

Curriculum Vitae

Personal Data

Name:	Stephan M. Winkler
Address:	Semleitnerweg 76 A-4111 Walding, Austria e-mail: stephan.winkler@heuristiclab.com
Dates of birth:	Linz, October 26th, 1980
Nationality:	Austrian
Marital status:	Unmarried
Position:	Research assistant Translational Research Programm project L284-N04: “GP-Based Techniques for the Design of Virtual Sensors”, executed by: - Upper Austrian University of Applied Sciences, Research Center Hagenberg - Johannes Kepler University Linz, Institute for Design and Control of Mechatronical Systems

Education

1985 - 1990	Elementary schools in Newark, Delaware, USA and Linz, Austria
1990 - 1998	Secondary school, high school: Bischöfliches Gymnasium Kollegium Petrinum, Linz, Austria
1999/10 - 2004/09	Studies in computer science Johannes Kepler University, Linz, Austria
2004/10 - 2007/02	PhD studies in engineering sciences Johannes Kepler University, Linz, Austria

Professional Career

1999	Programmer for Programmierfabrik GmbH, Hagenberg, Austria
since 2000	Programmer and IT counselor for Altenbetreuungsschule des Landes Oberösterreich, Linz, Österreich
2003 - 2004	Tutor for computer science at Johannes Kepler Universität, Linz, Austria
2004/10 - 2006/01	Research assistant at the Institute for Design and Control of Mechatronical Systems, Johannes Kepler Universität, Linz, Austria
2005/02 - 2006/03	Junior researcher at Linz Competence Center in Mechatronics (LCM)
2005/03 - 2005/06	Lecturer at Johannes Kepler Universität Courses in *Mechatronics*: Analysis of Mechatronical Systems
since 2006/02	Research assistant, Translational Research Program project L284-N04 “GP-Based Techniques for the Design of Virtual Sensors”, funded by the Austrian Science Fund (FWF)
since 2006/02	Research assistant at Research Center Hagenberg, Upper Austrian University of Applied Sciences (FH OÖ)
since 2006/10	Lecturer at Upper Austrian University of Applied Sciences, Campus Hagenberg Courses in *Software Engineering*, *Information Engineering and -Management*: Modeling and simulation, generative programming, software project engineering Courses in *Bioinformatics*: Software project engineering

Military Service

1998/08 - 1999/03	PzB 10, Kopal Kaserne, St. Pölten - Spratzern, Austria

Awards

2002, 2003	Scholarship of the technical faculty at the Johannes Kepler University for excellent merits in studies in the years 2002 and 2003.
2004/04	Best paper award of EMCSR 2004 for the contribution “Identifying Nonlinear Model Structures Using Genetic Programming Techniques” in the session “Theory and Applications of Artificial Intelligence”
2005/07	Best paper award of WMSCI 2005 for the contribution “Solving Multiclass Classification Problems by Genetic Programming” in the session “Management Information Systems”

List of Publications

2004:

[1] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Identifying nonlinear model structures using genetic programming techniques. *Cybernetics and Systems 2004*, pp. 689–694. Austrian Society for Cybernetic Studies. 2004.

[2] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. New methods for the identification of nonlinear model structures based upon genetic programming techniques. *Proceedings of the 15th International Conference on Systems Science.* Oficyna Wydawnicza Politechniki Wrocławskiej. 2004.

[3] Stephan Winkler. *Identifying Nonlinear Model Structures By Genetic Programming.* Diploma thesis. Institute of Systems Theory and Simulation, Johannes Kepler University Linz, Austria, 2004.

2005:

[4] Luigi del Re, Peter Langthaler, Christian Furtmüller, Stephan Winkler, and Michael Affenzeller. NOx virtual sensor based on structure identification and global optimization. *Proceedings of SAE World Congress 2005*, Detroit, paper number 2005-01-0050. 2005.

[5] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Solving multiclass classification problems by genetic programming. *Proceedings of The 9th World Multi-Conference on Systemics, Cybernetics and Informatics SCI 2005*, vol. 1, pp. 48-53. International Institute of Informatics and Systemics, 2005.

[6] Michael Affenzeller, Stefan Wagner, and Stephan Winkler. GA selection revisited from an ES-driven point of view. *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach*, LNCS 3562, pp. 262-271. Springer, 2005.

[7] Michael Affenzeller, Stefan Wagner, and Stephan Winkler. Goal-oriented preservation of essential genetic information by offspring selection. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) 2005*, vol. 2, pp. 1595-1596. The Association for Computing Machinery (ACM), 2005.

[8] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Genetic programming based model structure identification using on-line system data. *Proceedings of Conceptual Modeling and Simulation Conference CMS 2005.* 2005.

[9] Stephan Winkler, Hajrudin Efendic, Michael Affenzeller, Luigi del Re, and Stefan Wagner. On-line modeling based on genetic programming. *Proceedings of the 1st International Workshop on Automatic Learning and Real-Time (ALaRT'05)*, pp.

119-130. Institute of Real-Time Learning Systems, University Siegen, Germany. 2005.

[10] Daniel Alberer, Luigi del Re, Stephan Winkler, and Peter Langthaler. Virtual sensor design of particulate and nitric oxide emissions in a DI diesel engine. *Proceedings of the 7th International Conference on Engines for Automobile ICE 2005.* 2005.

[11] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. A genetic programming based tool for supporting bioinformatical classification problems. *Proceedings of the FH Science Day 2005*, pp. 3-10. Shaker Verlag, 2005.

[12] Peter Langthaler, Daniel Alberer, Stephan Winkler, and Luigi del Re. Design eines virtuellen Sensors für Partikelmessung am Dieselmotor. *Proceedings of the 14th Styrian Seminar on Control Engineering and Prozess Automation*, pp. 71-87. 2005.

[13] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. New methods for the identification of nonlinear model structures based upon genetic programming techniques. *Journal of Systems Science*, vol. 31, nr. 1, pp. 5-13. Oficyna Wydawnicza Politechniki Wrocławskiej, 2005.

2006:

[14] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Automatic data based patient classification using genetic programming. *Cybernetics and Systems 2006*, vol. 1, pp. 251-256. Austrian Society for Cybernetic Studies, 2006.

[15] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Advances in applying genetic programming to machine learning, focussing on classification problems. *Proceedings of the 9th International Workshop on Nature Inspired Distributed Computing NIDISC '06, part of the Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium IPDPS 2006*, IEEE Catalog Number: 06TH8860, ISBN: 1-4244-0054-6, ISSN: 1530-2075, paper nr. NIDISC-012. IEEE, 2006.

[16] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Sets of receiver operating characteristic curves and their use in the evaluation of multi-class classification. *Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2006*, vol. 2, pp. 1601-1602. The Association for Computing Machinery (ACM), ISBN 1-59593-186-4, ACM Order Number 910060. 2006.

[17] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Using enhanced genetic programming techniques for evolving classifiers in the context of medical diagnosis - an empirical study. *Proceedings of the GECCO 2006 Workshop on Medical Applications of Genetic and Evolutionary Computation (MedGEC 2006)*, paper nr. WKSP115. The Association for Computing Machinery (ACM), 2006.

[18] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. HeuristicModeler: A multi-purpose evolutionary machine learning algorithm and its applications in medical data analysis. In: A. Bruzzone, A. Guasch, M. Piera, J. Rozenblit (ed.), *Proceedings of the International Mediterranean Modelling Multiconference I3M 2006*, pp. 629–634. Piera, LogiSim, Barcelona, Spain, 2006.

[19] Stephan Winkler, Hajrudin Efendic, and Luigi del Re. Quality pre-assessment in steel industry using data based estimators. In: S. Cierpisz, K. Miskiewicz, A. Heyduk (ed.), *Proceedings of the IFAC Workshop MMM'2006 on Automation in Mining, Mineral and Metal Industry*, pp. 185–190. International Federation for Automatic Control, 2006.

2007:

[20] Stephan Winkler, Hajrudin Efendic, Michael Affenzeller, Luigi del Re, and Stefan Wagner. On-line modeling based on genetic programming. *International Journal on Intelligent Systems Technologies and Applications*, vol. 2, NOs. 2/3, pp. 255-270. Inderscience Publishers, 2007.

[21] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Advanced genetic programming based machine learning. *Journal of Mathematical Modelling and Algorithms*, ISSN 1570-1166 (print), 1572-9214 (online), DOI 10.1007/s10852-007-9065-6. Springer Netherlands, 2007.

[22] Stefan Wagner, Stephan Winkler, Roland Braune, Gabriel Kronberger, Andreas Beham, and Michael Affenzeller. Benefits of plugin-based heuristic optimization software systems. *Computer Aided Systems Theory - EUROCAST 2007.* LNCS 4739, pp. 747–754. Springer, 2007.

[23] Michael Affenzeller, Stefan Wagner, and Stephan Winkler. Self-adaptive population size adjustment for genetic algorithms. *Computer Aided Systems Theory - EUROCAST 2007.* LNCS 4739, pp. 820–828. Springer, 2007.

[24] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Selection pressure driven sliding window genetic programming. *Computer Aided Systems Theory - EUROCAST 2007.* LNCS 4739, pp. 788–795. Springer, 2007.

[25] Stephan Winkler, Bernd Brandl, Michael Affenzeller, and Stefan Wagner. Zeitreihenanalyse von Finanzdaten unter Verwendung von erweiterten Methoden der Genetischen Programmierung. Accepted to be published in *Proceedings of the First Science Symposium of the Austrian Universities for Applied Sciences* (Erstes Forschungsforum der Österreichischen Fachhochschulen). 2007.

[26] Michael Affenzeller, Stefan Wagner, and Stephan Winkler. Aspects of adaptation in natural and artificial evolution. *Proceedings of the Genetic and Evolutionary*

Computation Conference 2007, pp. 2595–2602. The Association for Computing Machinery (ACM), 2007.

[27] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Variables diversity in systems identification based on extended genetic programming. *Proceedings of the 15th International Conference on Systems Science*, vol. 2, pp. 470–479. Oficyna Wydawnicza Politechniki Wrocławskiej. 2007.

[28] Michael Affenzeller, Gabriel Kronberger, Stephan Winkler, Mihaela Ionescu, and Stefan Wagner. Heuristic optimization methods for the tuning of input parameters of simulation models. *Proceedings of I3M 2007.*, pp. 278-283. DIPTEM Università di Genova, ISBN 88-900732-6-8. 2007.

[29] Stephan Winkler, Michael Affenzeller, Stefan Wagner, Gabriel Kronberger, and Andreas Beham. Evolutionäres Design von Virtuellen Sensoren. *Proceedings of the Industrial Symposium Mechatronics 2007 on Sensorics*, pp. 166-175. Clusterland Oberösterreich, ISBN 978-3-9502270-1-7. 2007.

2008:

[30] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Offspring selection and its effects on genetic propagation in genetic programming based system identification. In Robert Trappl, editor, *Cybernetics and Systems 2008*, vol. 2, pp. 549–554. Austrian Society for Cybernetic Studies, 2008.

[31] Andreas Beham, Stephan Winkler, Stefan Wagner, and Michael Affenzeller. A genetic programming approach to solve scheduling problems with parallel simulation. Accepted to be published in *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS08).* IEEE, 2008.

[32] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. On the reliability of nonlinear modeling using enhanced genetic programming techniques. Accepted to be published in *Proceedings of the Chaotic Modeling and Simulation International Conference CHAOS 2008.* 2008.

[33] Stephan Winkler, Michael Affenzeller, and Stefan Wagner. Fine-grained population diversity estimation for genetic programming based structure identification and the effects of enhanced selection concepts. Accepted to be published in *Proceedings of the Genetic and Evolutionary Computation Conference 2008.* The Association for Computing Machinery (ACM), 2008.

[34] Michael Affenzeller, Stephan Winkler, and Stefan Wagner. Evolutionary system identification: New algorithmic concepts and applications. Accepted to be published as chapter of *Advances in Evolutionary Algorithms.* I-Tech Publishing, ISBN 978-3-902613-32-5. 2008.